THE UNIVERSITY OF CHICAGO

PRINCIPLES AND TOOL SUPPORT FOR DETECTING LATENT BUGS IN MODERN SOFTWARE SYSTEMS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES,
IN CANDIDACY FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY

BOGDAN ALEXANDRU STOICA

CHICAGO, ILLINOIS

AUGUST 2025

To my wife, Cristina, our daughter, Olivia,

my family, and the rest of the gang.

“To do is to be.”—Aristotle

“To be is to do.”—René Descartes

“Doo be doo be doo...”—Francis “Frank” A. Sinatra

“Yabba-Dabba-Doo!”—Fred Flintstone

“Scooby-Dooby-Doo!”—Scoobert “Scooby” Doo

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

What an incredible, fantastic, overwhelming, wild journey this has been!

Nearly two months ago I passed my Ph.D. defense, the second-to-last (and, in all honesty, the most important) hurdle before I could call myself "doctor". The final step is wrapping up this manuscript. Much to my committee's likely dismay, I spent a great chunk of the past two months writing these acknowledgments. Turns out, distilling six years into a few pages is the hardest task of all.

At the risk of sounding cliché, this has been an equally beautiful and bumpy road. There were plenty of highs, a few definite lows, with the inevitable mix of successes and disappointments—and I enjoyed it all. This adventure actually began years ago, across the pond, in a small Swiss town on the shores of Lake Geneva. I hadn't the slightest clue it would lead me here, in the Second City, typing up this dissertation. Of course, none of it would have been possible without the unconditional, and at times, stubborn support of my family, friends, and mentors. So this one goes out to all of you!

I owe my deepest gratitude to my fantastic advisor, Shan Lu. Every page of this dissertation exists thanks to Shan's tireless guidance, incredible patience, and immense care. It always amazes me how she could offer relentless advice not only to improve my research but also to help me grow as both a researcher and a person. It is truly rare to have a mentor who provides such close support while still allowing the space and independence needed to flourish. I graduate knowing I gained a trusted mentor, a caring advocate, and a treasured friend. I can never repay her kindness and support; I can only hope to provide my own mentees with even a fraction of the dedication Shan has shown me.

My first, and greatest, stroke of good fortune was having an amazing advisor. The second, was an extraordinary circle of mentors who rooted for me throughout my Ph.D. I am extremely grateful to my committee members, Haryadi Gunawi, Kexin Pei, and Suman Nath; close collaborators Madan Musuvathi, Jonathan Mace, Yang Wang, and Tianyin Xu;

and trusted confidants Ben Zhao and Pedro Fonseca.

I learned so much from Har, from his confidence in approaching some of the most challenging research questions, to his relentless dedication to closely mentor even the most junior student, and his fierce energy as he drove projects over the finish line. Above all, he taught me to think bigger and bolder, always keeping the long-term outlook in mind.

Working with Kexin has been a privilege. Kind, supportive, and always available, he spent countless hours helping me find my bearings in a subfield, AI-for-code, which I was rather unprepared to tackle. In the process, he constantly showed me how to be a better mentor myself.

If not for Suman, the hardest technical pieces of this dissertation might have gone unresolved (and with them, the thesis itself). He was consistently available with advice and feedback, and without fail gave solid answers to the toughest challenges I faced.

Conversations with Madan taught me never to shy away from tough questions. Our numerous chats, especially those before I started grad school at UChicago, gave me the calm and confidence that everything would work out.

Jonathan pushed me to refine my research argumentation and raise my standards. His candid and incredibly useful feedback, particularly while working on our SOSP'24 paper proved invaluable and helped me grow immensely.

From day one, Yang created the optimal conditions for me to explore ideas just outside my comfort zone. His patience and guidance while I navigated unfamiliar territory are gifts I will cherish long after graduation.

Tianyin went above and beyond to make me feel part of his group long before I formally joined as a postdoc. I am already learning a lot from his bold, long-term research vision. Equally important, his dedication, hospitality, and kindness know no bounds. I look forward to the years ahead working side-by-side with him.

I am nearly certain I would never have been admitted to UChicago if Ben would not

have echoed Shan's support of my application. As my graduate studies progressed, his sage advice, disarming humor, and perspective on grad school, truly reshaped my sense of what a Ph.D. can be and what should come next.

Although Pedro and I have not yet collaborated on a research project, we kept in touch and checked in throughout my Ph.D. He always lend a thoughtful ear to listen and offered thoughtful advice on both career and research life, for which I am extremely grateful.

It took a third, and equally crucial, stroke of good fortune to set this Ph.D. in motion and on its course. I would not have started my grad studies without the incredible support, guidance, care, and determination of two exceptionally devoted mentors: Vikram Adve and Bernard Moret.

Vikram took me under his mentorship early on during my pre-UChicago years, with unwavering support. He went out on a limb for me in countless ways, offering patient, meticulous guidance that sharpened my research tastes and shaped my present career trajectory. He remained my staunchest supporter and played a decisive role in helping me jump-start my graduate studies.

All throughout my Master's studies and even after wrapping up, Bernard had my back. He offered me the rare chance to teach and mentor undergraduates when no one else would, and showed me incredible kindness and care, making me feel part of the LCBB family from the very first day. Always available with sound advice, he was my most enthusiastic supporter, fueling my resolve to take bold steps and apply to grad school.

I would like to express my deepest gratitude to the extended UCARE gang, past and present: Utsav Sethi, Chengcheng Wan, Junwen Yang, Yuhan Liu, Guangpu Li, Lefan Zhang, Shu Wang, Yuxi Chen, Haochen Pan, Chi Li, Ruidan Li, Rani Ayu Putri, Ray Andrew, Roy Huang, Daniar Kurniawan, Meng Wang, Cesar Stuardo, Jeffrey Lukman, Huaicheng Li, Weichen Li, and Jun Yang. I am especially grateful to Utsav for his steady advice and for always putting a positive spin on things. Together, we tackled one of the most daunting projects of our grad studies, and we would not have crossed the finish line without his unwavering drive and good humor during countless late-night sprints. I am equally grateful to Guangpu who was constantly available for a reassuring chat and looked out for me throughout my junior years. From him I learned not only concrete research techniques but also a broader, more philosophical view of the Ph.D. journey. I am very thankful to Jun for making the last year of my Ph.D. productive and engaging, and for helping me, alongside Kexin, learn the ropes of a subfield slightly outside my comfort zone.

I am profoundly grateful to our awesome support staff: Nita Yack, Devin Brooks-Davis, Nick Seamons, Marc Richardson, Colin Hudler, Shashi Dyamenahalli, Justin Laughlin, Kristin Czaplewski, Roberto Vale, Megan Woodward, Margaret Jaffey, Bob Bartlett, and Tom Dobes. My PhD journey was so smooth, in no small part, because of your prompt support, timely coordination, and handling last-minute requests with unfailing grace.

I am sincerely thankful to everyone in Tianyin's research group for their hospitality and for making me feel part of the team long before I officially joined. A huge shout-out to Yiming Su, Xudong Sun, Tyler Gu, Siyuan Chai, Jongyul Kim, Shuai Wang, Yinfang Chen, Cody Rivera, Cathy Cai, Wentao Zhang, Jackson Clark, and Jiyuan Zhang. I am incredibly thankful to Yiming. Yiming and I go way back, to his undergraduate days at UChicago, where we crunched through many daunting deadlines together. His trek up north to attend my thesis defense in person was a kindness I will remember fondly.

It takes a village to raise a Ph.D. graduate, and in my case that could not be more true. So, in no particular order, Monica Chiosa, Igor Zablotchi, Cristina Andrei, Catalina Husanu, Tanja Petricevic, Iulian Dragos, Ana Petricevic, Oana Balmau, Corina Stoenescu, Camilo Rojas, Florin Dinu, Marios Kogias, Andrea Miele, George Prekas, Cécile Bousquet, Qianqian Wang, Laura Piccinini, Adriana Gelbart, Stéphanie Gonvers, Ela Verman, Alexandra Mihaela Olteanu, Cristina Ghiurcuta, Sonia Duc, Mihai Dobrescu, Calin Iorgulescu, Danut Necula, Matej Pavlovič, Mia Primorac, Adrien Ghosn, Slobodan Mitrović, Stanko Novaković, Dan Salajan, George Popescu, Sylviane Dal Mas, and Tania Epars—from the bottom of my heart, thank you! Without you, this path would have been far lonelier and I would have been much more grumpier than usual. Having you around has made every bit of it worth the effort, and then some. Your friendship reminds me every day that success is more meaningful when the entire village celebrates.

However, none of this would have been possible without the unconditional support, patience, and love of my family.

I cannot put into words my gratitude to my parents, Marilena and Adrian, for nurturing my passion for computers since I was 8. They bought me my first laptop, took me to my first computer club, and spent hours printing grid paper so I could draw pixel-by-pixel characters and code them into silly video games. They took me, or rather allowed themselves to be dragged, to every conceivable computer-related contest and event, and continued to support me unconditionally even after I moved an ocean away.

To my brother, Dragos, thank you for your laid-back optimism, for always being encouraging and eager to hear about my projects. Whenever experiments failed or papers got rejected, you were there to remind me there is plenty of life beyond academia.

My cousin Radu has saved my back in more ways than I can remember. During my early research days he devoted countless hours teaching me how to write, present, and pitch my work, and he also spared me from homelessness by offering a place to stay while I wrapped up my Master's. I am also grateful to Alice and to my wonderful nieces, Margareta and Rosana, who, together with Radu, made every layover in Zurich worthwhile and fun.

A sincere thank-you to my in-laws, Monica, Iulian, and Andreea: your steady reassurance that everything will turn out great has kept me calm and grounded.

Yet, there is one person without whom this thesis would never have been written. My incredible wife, Cristina. Since she became my best friend over twelve years ago, she has been a mountain of calm, wisdom, and joy, listening patiently to my every story, wild idea, worry, and rant. She knows the ins and outs of my research, sometimes better than I do. If the reader thinks writing this dissertation was my smartest move, they could never be more wrong. The best decision I ever made was marrying her. Not only do we make a fantastic team, but recently we added one more team-member when our amazing daughter, Olivia, was born. A bundle of joy, laughter, and, occasionally, milk, who has made all this so, so much more worth it.

ABSTRACT

Modern software systems are designed for high availability, resilience, and performance. Their scale, however, makes them prone to latent, complex bugs that escape in-house testing. These defects—ranging from subtle concurrency issues to cross-component or cascading faults, to performance bottlenecks—typically surface under rare timing, unusual external events, excessive resource utilization, or unexpected workloads that existing tests seldom detect. Left unchecked, they trigger catastrophic failures, severe outages, and serious data corruption, at multi-billion-dollar remediation costs.

As the following chapters will show, current testing infrastructure and bug-finding tools, while useful, often fail to detect such bugs for three key reasons. First, techniques that explore the full state space or apply expensive program analysis do not scale to the complexity of modern systems. Second, lightweight approaches, in contrast, require a prohibitive number of test executions, handcrafted annotations or specialized inputs. Third, the vast majority of these techniques focus on low-level symptoms, thus routinely missing "higher order" bugs that surface from the interaction of code, operational policies, and workloads.

To address these limitations, this dissertation proposes a unified paradigm based on two key principles that improve both bug detection coverage and practicality. First, techniques presented in the next chapters *actively perturb* the program's execution—through thread delays (timing), transient faults (fault injection), or adversarial inputs (test generation)—to exercise rare, faulty code paths. These perturbations are guided by semantic analysis derived from lightweight static/dynamic analysis, execution feedback, and AI-assisted heuristics, thereby trading soundness for practicality. Second, techniques described in this dissertation repurpose the extensive test suites that deploy with modern software systems, turning existing tests into bug triggers, without forcing software engineers to craft new tests, which is a tedious and time-consuming task.

We implement these principles by designing techniques that target bugs at three levels

of abstraction: (program) statement, mechanism, and module.

Chapter 3 explores bug-finding at the *(program) statement level*, targeting the interaction of individual program statements whose correctness depend on their relative order, the value of their operands or their side-effects. This chapter introduces WAFFLE, a tool that triggers concurrent memory ordering violations (MEMORDER bugs) which occur when an object is accessed before initialization or after disposal. WAFFLE profiles one injection-free execution by running the program's test suite once to capture object lifetimes and thread parent-child relations. It then injects carefully timed delays at a few candidate memory access sites by running the test suite again. WAFFLE detects 12 known and 6 unknown bugs in popular C# open source distributed applications, with moderate run-time overhead. This chapter is based, in part, on [210].

Chapter 4 explores bug-finding at the *mechanism level*, where detection is elevated to cohesive control code structures that enforce reusable policies (e.g., retry). Here, correctness depends on the implementation itself plus the state, policies, and execution limits (e.g., back-off) it operates on. This chapter presents WASABI, a bug detection tool that uncovers bugs in retry mechanisms. WASABI combines lightweight control-flow analysis with the semantic reasoning capabilities of large language models (LLMs), to locate retry code structures that traditional code analysis miss. It then executes a subset of the application's test suite, injecting transient exceptions at retry locations identified previously. Finally, it observes the application behavior and applies specialized test oracles to identify retry bugs. WASABI uncovers over 100 retry bugs across eight widely used Java distributed applications. This chapter is based in part on [211] and focuses on WASABI's active testing and fault injection components. The interested reader can consult [211] to learn about WASABI's LLM-driven static bug detection capabilities (Sections 2, 3.2, 4.1, 4.3, and 4.4).

Chapter 5 explores bug-finding at the *module level*, where the unit of analysis is an entire algorithm, service, or component. Here, correctness is determined by observable execution properties (e.g., latency, CPU or memory utilization) and the target program's interaction with workloads or its external environment. This chapter introduces WEDGE, a tool that aims to uncover cost-amplifying execution paths (i.e., paths whose resource usage grows super-linearly) that functional tests often overlook. WEDGE first profiles existing tests and selects two traces whose running times diverge on similar inputs. It then prompts an LLM to synthesize *performance-characterizing constraints*, boolean predicates that aim to explain why the code structure and semantics might determine the differences observed in the two contrastive execution profiles. Finally, it inserts these predicates as extra branches in the original code and leverages a coverage-guided fuzzer to find inputs that satisfying them. Overall, WEDGE increases average execution cost by 84% compared with state-of-the-art AI-guided performance (stress) test generation tools on several popular C/C++ benchmarks. This chapter is based, in part, on [236] and focuses on how WEDGE leverages LLMs and contrastive reasoning on execution traces to infer performance-characterizing constraints. The interested reader can consult [236] for a more elaborate discussion about how WEDGE and its extensions are engineered (Sections 3 and 4.2; Appendixes A and B).

Collectively, these tools demonstrate that guided perturbation combined with repurposing existing test suites are a scalable, general strategy for surfacing difficult-to-detect latent, complex bugs in modern software systems.

CHAPTER 1
INTRODUCTION

Modern software systems consist of distributed, interconnected components built for high availability, resilience, and performance. As these systems scale, more bugs evade routine in-house testing and surface after deployment. Recent studies report a growing class of latent, complex defects escaping in production that are highly disruptive and hard to detect [80, 91, 143, 217]. These bugs frustrate traditional testing infrastructure, manifesting under rare and specific conditions, typically involving infrequent timing occurrences, unforeseen external events, excessive resource utilization, or unexpected workloads. Some occur only when the system nears operational capacity (e.g., highly concurrent, memory-intensive executions), while others are obscured by automated failover or resilience mechanisms (e.g., task retry, cancellation, or throttling). Challenging still, some are symptom-silent, not generating any obvious effects, like exceptions or crashes. Left unresolved, these latent defects have a significant impact on the system's reliability, availability, and performance, causing severe outages [16, 46, 67, 128], catastrophic failures [22, 73, 224, 234], and incurring billions of dollars in remedy costs [50, 127, 129, 207].

Software engineers build modern software systems with reliability, fault tolerance, and redundancy in mind. They therefore deploy and operate such systems with an extensive array of testing procedures meant to harden them against failures. Yet even comprehensive test suites designed for a wide array of operation and misoperation scenarios rarely recreate the precise timing, external conditions, or computational constraints required to manifest those difficult-to-detect defects.

To counter this, a significant amount of research aims to detect, troubleshoot and fix latent, complex bugs in modern, large-scale systems. Unfortunately, while useful and worthwhile, these techniques suffer from three intertwined shortcomings that make then unsuitable or less effective against such defects.

First, there is a persistent tension between scalability, automation, and bug coverage. Many traditional bug-finding tools either rely on expensive program analysis or resort to brute-force testing, neither strategy well-suited for modern systems (e.g., emerging Cloud-native infrastructures). Some of these techniques require to explore a large space of possible states and execution modes [33, 76, 154, 173, 191, 196, 201, 246], or systematical analyze large code bases with algorithms that scale poorly with the program size [82, 104, 198, 245]. Others, rely on a much lightweight code analysis and in contrast resort to randomized testing or fault injection [4, 27, 29, 42, 77, 81, 82, 100, 104, 133, 154, 171, 198, 245, 246]. While these tool achieve better scalability, they often require prohibitive execution budgets or need a significant number of trials to find bugs [143].

Second, many of these tools rely on manual annotations [9, 25, 44, 69, 98, 134, 151, 193, 235, 240, 243] or exploit specific characteristics of the system-under-test [8, 10, 89, 118, 126, 165, 167, 216] to steer program execution towards faulty states. While effective, this specialization also makes such tools brittle, as they often struggle to generalize to modern, complex software and sometimes cannot adapt without tedious manual intervention [80].

Third, with only a few recent exceptions [40, 202, 229], the vast majority of bug-finding tools still focuses on low-level symptoms, missing higher order bugs that emerge only when code syntax and semantics are considered along side operational policies [105](e.g., whether to retry a transient error, cancel a stalled task, etc.). Many of these tools check individual program statements to identify logic bugs or, in the case of detecting performance defects, either rely on specialized, manually crafted tests or on automatically generated workloads that exhibit mostly generic, commonly known patterns [80, 91, 137, 155]. Recently empirical studies show that a significant fraction of latent bugs in modern software systems surface from particular complex interactions across code fragments, modules, workloads, and operational policies [80, 146, 155, 217]. While promising, tools that are more aware of those higher order bugs still rely on considerable manual effort to identify specific failure patterns [202], or focus on domain-specific failure modes to limit the complexity of the underlying program analysis [40, 229].

1.1 Dissertation contributions

This dissertation shows that guided perturbation of existing test suites, leveraging on feedback-driven timing delays, AI-guided fault injection, and AI-directed input mutation, can more effectively detect correctness bugs and performance inefficiencies that escape traditional program analysis and ad-hoc testing.

Techniques developed in the next few chapters deliberately steer the execution of existing tests toward rare failure states by injecting timing delays, transient faults, and adversarial inputs that attempt to emulate production-time anomalies. Instead of relying on heavy-weight, sound code analysis, this work relaxes specific analysis stages and replaces them with lightweight, unsound heuristics informed by program semantics, run-time feedback, and AI-assisted code comprehension, trading soundness for scalable and practical bug detection. The tools described in this dissertation leverage the fact that today's systems already deploy with hundreds (sometimes, thousands) of unit, functionality and integration tests that encode realistic usage patterns. They take full advantage of this and convert existing tests into effective bug triggers with minimal development costs, instead of relying on software engineers to craft specialized scenarios—a time-consuming effort.

This dissertation explores bug-finding at three abstraction levels: (program) statement, mechanism, and module. The next chapters, thus, aim to address issues spanning the full spectrum of latent, complex bugs encountered by large-scale, modern systems.

Chapter 3 explores bug-finding at the *statement level*, by targeting individual memory and control-flow operations where perturbing the execution can corrupt program state. Statement level analysis treats a target program as a sequence of atomic events (e.g, load/store operations, conditional branches, synchronization operations, etc.) whose relative order and operands fully determine local correctness. At this granularity, a bug occurs when two or more such statements execute in an unintended temporal order or with inconsistent values (e.g., harmful data races, deadlocks, buffer overflows, `NULL`-pointer exceptions). This chapter shows how active delay injection can be redesigned to be lightweight, practical, and effective at exposing memory ordering violations (MemOrder bugs).

Chapter 4 examines bug-finding at the *mechanism level*. Mechanism level analysis raises the abstraction unit of analysis from a single statement to a cohesive control code structure that implements a reusable policy, such as a task re-execution loop (retry), access control workflow (throttling), or a more generic state-machine logic. At this granularity, incorrect behavior is rarely attributed to a single line of code. Instead it emerges from the aggregate evolution of the overall implementation of the mechanism, along with its state, decision rules, and possible execution limits on control and data paths. Code analysis must first discover where and how the mechanism is implemented, often spanning multiple methods or files, and then reason about its run-time properties (e.g., bounded retries, exponential back-off, or idempotency). Whether a bug occurred is determined based on higher-level guarantees which combine concrete implementation with code semantics and execution policies, rather than relying solely on localized, statement-level properties. This chapter, thus, introduces techniques to expose bugs in the retry logic of modern systems, by using AI-informed program analysis to extract code structure information, combining it with timing and policy-level behavior to formulate effective fault injection strategies to trigger those bugs.

Chapter 5 explores bug-finding at the *module level*, where the analysis takes a more holistic view of the larger component. Module level analysis considers an entire algorithm, service, or workflow logic as the analysis unit and evaluates behavior through observable properties such as latency, CPU cycles, memory and I/O utilization, network traffic, and the integrity of end-to-end results. A bug at this granularity can manifest as an emergent correctness or performance issue that surfaces only when a rarely exercised state/control-flow boundary is crossed. The key challenge of detecting those bugs is semantic: the trigger condition is expressed in terms of an external input or interaction, while the symptom appears deeply nested within the module's implementation. Worse still, simple pass/fail bug oracles are often inadequate. A module level bug oracle may need to compare resource cost against a baseline and validate overall correctness before labeling the execution as faulty. This chapter demonstrates how to infer performance-related code properties and leverage them to synthesize more efficient stress tests, by combining AI-guided approaches with traditional input synthesis techniques like fuzzing to generate tests that steer execution towards more cost-amplifying (i.e., hot) paths.

Finally, this dissertation explores how AI-assisted reasoning about program semantics can support bug detection, and hopes to encourage future work exploring AI-guided approaches at every stage of the software quality lifecycle, from triggering hidden failures and performance bottlenecks to pinpointing, diagnosing, and ultimately fixing them. Chapters 4 and 5, in particular, show how traditional program analysis and bug detection techniques can rely on AI-assisted approaches to speed up interpreting code structure, reasoning about software engineering intent (e.g., is this retry functionality or something else?), and inferring which program statements, code structures, or workflow logic needs to be investigated.

1.2 Dissertation organization

This dissertation is organized into three main parts.

(Part I) Analysis at the statement level: schedule-guided memory ordering bugs detection. Concurrency bugs are notoriously difficult to trigger, capture, and diagnose. In particular, memory ordering (MemOrder) bugs, a type of concurrency defect caused by unsynchronized access to an object before initialization or after deallocation, are both common and severe. These can lead to memory corruption, crashes, and serious security vulnerabilities.

Traditional concurrency bug-finding tools either rely on expensive static analysis of synchronization primitives, or random scheduling, which makes them impracticable. Fortunately, recent delay injection techniques that target atomicity violations—the dominant type of concurrency issues [162]—demonstrate that it is possible to implement effective detection policies by relying on injection feedback, program behavior, and lightweight instrumentation [143]. However such approaches struggle exposing MemOrder bugs due to the following fundamental differences. Atomicity violations require overlapping execution windows, whereas order violations, and consequently, MemOrder bugs need operations to execute in reverse order rather than to simply overall, leading to many more potential injection points. In fact, our initial direct adaptation of the key ideas behind Tsvd [143] required us to probe 10× more program locations in order to surface MemOrder bugs, yet triggering only half the MemOrder defects present in our benchmarks.

Inspired by these early attempts we propose Waffle, a tool that breaks down active delay injection into four fundamental design points—where, how, when, and for how long to inject delays—and explores new trade-offs for each.

Waffle first runs the program without delays to identify candidate injection points through a lightweight analysis that infers thread fork/join relationships, focusing on memory accesses that might lead to MemOrder bugs. In subsequent runs, it injects delays at these locations using heuristics to minimize interference and overhead. The key insight behind this work is systematically characterizing the architecture of active delay injection, tailoring the four key design points mentioned earlier to match the manifestation patterns of MemOrder bugs.

Waffle is evaluated on 11 real-world, popular C# projects, it uncovered 18 MemOrder bugs, including six previously unknown, and reliably reproduced 15 out of 18 within just two runs, all at only a 2.5× slowdown over uninstrumented runs, significantly outperforming both multi-run sampling approaches and heavyweight synchronization analyses.

(Part II) Analysis at the mechanism level: semantic fault injection for triggering bugs in recovery logic. In contrast to bugs that occur at the program statement level, some hard-to-detect bugs happen at the mechanism level and involve higher order system behaviors such as re-execution, timing, throughput, and fault-handling policies. Retry bugs, for example, often manifest under rare, hard-to-replicate conditions and can lead to performance degradation, increased load, or cascading system failures. Traditional program analysis tools struggle to detect these issues because retry logic does not follow consistent patterns, frequently using complex structures like asynchronous task re-enqueuing or state machines. Only 55% of retry logic appears as loop structures, while 45% is implemented as non-loop constructs [211]. Additionally, existing unit tests seldom expose retry bugs, as they are not designed to simulate transient errors or explore nested retry iterations.

Despite being the last line of defense against myriad system failures, retry mechanisms, when faulty, can cause serious or catastrophic failures. A study [211] of 70 real-world retry incidents across eight popular Java systems revealed that bugs split evenly among three root causes, namely wrong decision to retry (36%, denoted as "IF" bugs), incorrect timing between attempts or total number of repetitions (33%, denoted as "WHEN"), and wrong implementation of the mechanism itself (31%, denoted as "HOW").

To address these challenges, we propose a new bug-finding technique that combines static analysis, dynamic fault injection, and large language models. Specifically, we leverage complementary traditional control flow analysis combined with code comprehension performed by an LLM to identify where retry logic is implemented in the target codebase. Then we use fault injection and software testing to trigger bugs in those code structures.

We implement these idea in a toolkit called Wasabi. At its core, Wasabi transforms existing unit tests into effective bug triggers by forcing them to exercise retry paths under rare, transient errors that can expose retry issues. In particular, Wasabi inserts these transient faults the retry sites identified earlier through LLM-guided static analysis and checks for missing retry limits, absent back-off delays, or improper exception handling.

In parallel, WASABI relies on LLM-assisted static analysis inspects under-covered code regions for ad-hoc retry code structures and labels likely faulty retry, catching bugs that existing tests do not cover and thus cannot trigger [211].

WASABI can thus systematically detect IF, WHEN, and HOW retry bug patterns by repurposing existing tests and without requiring software engineers or system operators to write specialized ones. WASABI found 42 bugs by actively injecting transient errors (Chapter 4) and an additional 87 bugs through LLM-enhanced static analysis. Interested readers can check out the extended version of this work [211] for more details on the static analysis workflow of WASABI.

(Part III) Analysis at the module level: predicate-directed test synthesis for improving code efficiency Many performance bugs occur only when specific inputs drive the system into an expensive (hot), cost-amplifying execution path. Conventional unit tests rarely supply those stress inputs, and typical code analysis on which performance profilers rely can only measure the workloads they are provided with, leaving software engineers unaware of expensive paths. Generating those types of performance-stressing inputs automatically is challenging for two key reasons: (i) such detection tools must isolate where the slowdown originates, and (ii) create concrete inputs that actually satisfy the triggering conditions.

One difficulty is bridging the gap between internal performance inefficiency conditions and the external inputs that produce them. On the one hand, traditional software fuzzing tools (e.g., PerfFuzz [137], SlowFuzz [187], FuzzFactory [179], etc.) may discover performance-stressing inputs with the help of a signal such as user-provided cost proxies or specialized annotations indicates to the mutation engine that the execution entered an a cost-amplifying path. On the other hand, while LLMs have been successfully prompted to generate longer or larger inputs, semantic triggers often matter more than length/scale alone.

As a first step towards effectively generating performance-stressing tests, we present Wedge, an active testing framework that uses LLMs to reason about performance at the *module level* and use those insights to craft inputs that trigger performance bottlenecks. Wedge operates in two main phases. First, it performs contrastive profiling on a target module by comparing resource-usage metrics across different executions and inputs. This step pinpoints cost-amplifying code paths that cause resource saturation (e.g., unusually high CPU or memory use). Second, Wedge instruments the original code with these inferred constraints in the form of small predicate checkers and uses a coverage-guided fuzzer to synthesize inputs that satisfy those constraints One current limitation of this approach is that as code size grows, LLMs face challenges due to restrictions on context window sizes. Consequently, Wedge currently targets smaller modules spanning only several methods., so that the relevant code fits within the LLM's prompt window.

Wedge synthesizes more effective performance-stressing tests than both the default test suites, state-of-the-art AI-for-code benchmarks, and most recent AI-informed stress test generation tools, slowing down program execution by 84% on aberage. Our tests also help measure performance improvements claimed by AI- and feedback-guided optimization techniques more fairly, increasing the gap between optimized and non-optimized code execution by 24%-149% in terms of number of instructions executed (instruction count) and by 5% to 27% in terms of physical running time.

Despite this limitation, Wedge constitutes a concrete first step towards AI-guided performance analysis: it demonstrates that profiling, program analysis, and LLM-guided fuzzing can be combined to actively trigger such defects, laying the groundwork for more scalable performance-focused testing tools in future work.

1.3 Dissertation scope

This dissertation focuses on *bug detection* and does not explore bug diagnosis or code repair. Bug detection is the process of confirming the presence of a defect. In contrast, bug diagnosis (root-cause analysis) seeks to explain why it occurs, while code repair attempts to fix it. Bug detection is often the first step in the software quality lifecycle, the precursor to diagnosis and repair.

This dissertation investigates *software* and primarily focuses on correctness (Chapter 3 and Chapter 4), and performance (Chapter 5) defects at the expense of other important classes such as security, misconfiguration, or misoperation bugs. Furthermore, this dissertation investigates bugs that occur in a system's *core modules*. It does not address bugs in "external" code like those present in ancillary components (e.g., test harnesses), configuration and deployment code, or orchestration infrastructure.

This dissertation examines how *existing tests* can be repurposed *as-is* or *automatically adapted* into effective bug triggers in a mechanized fashion. It assumes software engineers will not (manually) craft specialized tests or workloads to expose the specific bugs discussed in later chapters.

Lastly, this dissertation advocates embracing *controlled imprecision* as a paradigm to tackle complex bugs in modern software systems, by trading soundness and precision for bug-finding practicality. Specifically, it advocates for exploring how AI-guided approaches can complement, rather than replace traditional software reliability techniques such as code analysis, testing, and generation, throughout the software quality lifecycle.

CHAPTER 2
BACKGROUND AND PREVIOUS WORK

This chapter surveys a fraction of the prior work, most relevant to the challenges, techniques, and tools proposed in this dissertation. We postpone an in-depth discussion on how the present work builds upon, improves, and expands the papers described below, for later chapters (§3.7.3, §4.4.3, and §5.2.1, respectively).

2.1 Empirical Studies for Failures in Modern Software Systems

A wide range of papers is dedicated to studying failures in cloud and distributed systems [28, 80, 90, 93, 105, 117, 147, 155, 202, 217, 248]. Some works introduce new taxonomies for bugs in cloud and distributed systems [80, 90, 155], while others focus on new or understudied classes of bugs like metastable failures [28, 105], cross-system defects [217], upgrading bugs [248], and performance degradation issues [93, 117, 147].

2.1.1 Outages in modern software systems

Several studies categorize outages occurring in modern, large-scale systems. Gunawi et al [90] examine over 3,000 issues from six popular open-source systems, and identify new bug categories specific to Cloud environments, such as data consistency errors, scalability issues, or topology-related defects. Many of these bugs could could cascade across multiple nodes or the entire cluster, causing widespread failure. In a follow-up study [91], they investigated causes of Cloud outages by investigating close to 600 publicly available reports. The most common outage causes were software bugs, upgrade mistakes, and misconfigurations, followed by network issues and overload conditions when systems reach their operational capacity. Similarly, Liu et al [155] studied hundreds of high-severity production incidents from Microsoft Azure, and reported that the most common causes are software bugs (nearly half), faulty failure handling, concurrency bugs, and inconsistent engineering assumptions across different components (e.g., data format issues, misconfiguration, over-/under-provision).

2.1.2 Faulty upgrades, cross-system failures, and task interaction bugs

As modern systems grow in complexity, many failures originate from the interaction between components, rather than isolated bugs in one module. Several works provide a detailed taxonomy of faulty upgrades, cross-system issues, or task interaction failures that happen at scale.

For example, software upgrades and multi-system integration are becoming more common [248]. Common root causes included data format incompatibilities between versions, persistent state mismatches, and errors in upgrade scripts or ordering. Authors also note that nearly one-third of severe outages studied by other researchers had an upgrade mistake at their core.

Failures can also occur across system boundaries, leading to incidents where no single component is faulty, but mismatches or hidden assumptions between systems lead to errors. Tang et al [217] analyzed over 100 failure reports in open-source software and close to a dozen incidents in public Cloud systems. Overall, almost 37% of the issues in open-source reports involved interaction between multiple systems, and about 20% of the cloud outages had cross-system root causes.

2.1.3 Performance degradation

Another well studied set of incidents focuses on failures that degrade performance or availability without necessarily crashing the system.

One of the main causes of performance degradation are performance bugs. For example, Li et al [147] investigates 99 real-world performance bugs found in popular open-source distributed systems. Their study reveals that common root causes for performance bugs are related to improper resource utilization, “blocking” defects that cascade and significantly delay normal operation, synchronization issues, use of sub-optimal algorithms, and misconfigurations. These findings further confirm earlier studies that analyze cascading and blocking defects [146], as well as algorithmic inefficiencies such as inefficient computation, API misuse, and incorrect caching or batching [117].

Gunawi et al [93] studies fail-slow incidents, when a faulty component remains operational in a reduced state instead of failing. Their study focused on fail-slow issues affective hardware, and revealed that essentially every type of hardware (e.g., SSDs, HDDs, memory, network cards, power supplies), can exhibit limping behavior, degrading in performance by orders of magnitude without failing completely.

Recently, researchers identified another class of issues characterized by a degraded state that persists even after the initial bug trigger is removed, called metastable failures [28, 105]. The most common root causes were fault retry logic (over 50%), computationally expensive or verbose error handling, work amplification, resource exhaustion, and lock contention.

2.2 Fault Injection Tools

2.2.1 Domain-specific fault injection

Numerous papers triggers bugs by injects domain-specific faults like network partitions, disk errors, database read/write inconsistencies [8, 10, 19, 89, 118, 126, 165, 166, 167, 229]. For example, Ju et al [118] designed a fault injection tool triggering bugs in OpenStack services by injects faults such as process crashes, message loss, and network partitions into its different components. Pace [8] is a tool focusing on failure scenarios where all components of a sub-system (e.g., storage) crash at once, a concept coined as *correlated crashes*. Elle [126] and LFI [10] are designed to expose bugs in distributed data management systems, by injecting faults based on transaction history and data lineage. FATE and DESTINI are part of the same cloud recovery testing framework that systematically explores multiple failure combinations and uses available software specifications to detect incorrect recovery behaviors [89].

Legolas [229] focuses on partial failures in distributed systems, also known as *gray failures*. It strategically injects faults at the application level by throwing exceptions to maximize the system's abstract state coverage. It also relies on existing system-crash and gray-failure test oracles to determine whether a bug has occurred, and relies on integration tests to steer the execution towards a failure.

2.2.2 Component-specific fault injection

Other works targets more general failures in specific system components or protocols such as cluster controllers [216] Cloud operators [87], or REST API logic [40, 169].

Recently, Sun et al [216] introduced Sieve, an automated testing tool for cluster controllers (e.g., Kubernetes). Sieve perturbs the execution of a controller by temporary delaying cluster events or injecting "fake" events that did not execute, and compares the perturbed execution with a regular run to identify inconsistent behavior.

Gu et al [87] designed Acto, an automated tool to test cloud system operators (e.g., Kubernetes). Acto automatically generates sequences of operator events that steer an operator from an initial to a desired state, trying to simulate scenarios where the system running atop does not reach the desired state, or the operator itself cannot successfully recover from an failure state.

Rainmaker [40] targets bugs triggered by transient errors in applications interacting with cloud services via REST APIs., focusing on transient errors like service unavailability and timeouts. Rainmaker intercepts API calls at the HTTP layer, enabling fine-grained fault injection without modifying the application code. Its fault injection follows a taxonomy of HTTP-specific bug patterns and code- coverage metrics. Its oracles use existing test assertions to detect issues such as inconsistent exceptions, silent state divergence, and unhandled transient errors.

More broadly, Filibuster [169] runs the standard tests a microservice comes with and injects a wide variety of faults to simulate partial-failure scenarios. The tool can inject faults at the inter-communication layer (i.e., HTTP error codes), common exceptions thrown by database clients, and data corruption scenarios.

2.2.3 Manifestation-specific fault injection

Several papers focus on specific failure scenarios like crash- recovery [89, 118, 161] and crash-consistency [8, 131, 170, 189], which are typically triggered by non-recoverable errors.

For instance, CrashTuner [161] is a testing tool aimed at finding crash-recovery bugs, by automatically identifying in-memory variables that contain *meta-information* (e.g., node count, current leader machine ID, etc.). It then inserts stubs right before reading or right after updating those variable and checks the system behavior at one of those points.

Chipmunk [131] is designed to tackle crash-consistency faults in persistent memory (PM) file systems. The tool instruments key PM `write()` functions with handlers using Linux kernel probes, intercepts when a PM file system is flushing data to memory, and selectively injects crashes at those program points when a system call is currently executed.

CrashMonkey [170] is a tool that triggers crash consistency bugs on file systems. The tool automatically generates all combinations of up to n (for a small value) file system operations (e.g., `fsync()`, `rename()`), and simulate the effects of a crash before an arbitrary operation in this sequence.

2.2.4 Fault injection with manual support

Other works require users to specify or customize injection rules [9, 44, 98, 243] or test oracles [34, 37]. For example, *Gremlin* [98] is a framework that allows software engineers to test the resilience of their microservices, by having them manually specify a service-specific fault injection policy and declare custom assertions to validate expected system behavior.

2.3 Fuzzing Tools

A wide array of works aim to synthesize bug-triggering inputs using fuzzing. Some fuzzing tools focus on generic input generation [17, 26, 74, 107, 138, 199, 223, 238], while others specialize on kernel bugs [82, 83, 112, 197, 215, 233], network defects [11, 14, 21, 125, 140], hardware failures [119, 208, 221], concurrency bugs [41, 82, 83, 112, 113, 152, 192, 195], and performance issues [30, 36, 137, 179, 187, 244], to just name a few. We only focus our attention on fuzzing tools geared towards concurrency and performance bug, as these are most relevant to the techniques presented in this dissertation.

2.3.1 Fuzzing for concurrency bugs

Razzer [113] leverages fuzzing to effectively generate system-call sequences whose concurrent execution can help expose concurrency bugs. ConAFL [152] combines traditional timing perturbation with input fuzzing. Specifically, it inserts code stubs that adjust thread-schedule priority right after thread creation and around potential bug locations[1], similar to previous concurrency-bug detection tools [198, 246]. It then applies AFL [238] on the modified application to increase the fuzzer's capability of exposing concurrency bugs.

2.3.2 Fuzzing for performance bugs

SlowFuzz [187] and PerfFuzz [137] are feedback-driven fuzzers that search for inputs causing extremely long execution times or high resource utilization. PerfFuzz in particular uses multi-dimensional feedback to independently maximize execution counts for many program

1. To identify these locations, ConAFL uses static analysis that cannot scale beyond ten thousand lines of code.

locations, thus finding a variety of inputs that exercise different cost-amplifying (hot, frequently exeucted) paths. FuzzFactory [179] generalizes this idea by allowing software engineers to define custom performance-oriented feedback metrics and integrates them into a fuzzing framework.

2.4 Bugs in Error Handling and Recovery Logic

Bugs in error handling and recovery logic are still one of the main root causes for production failures in modern large-scale systems [90, 155, 237].

2.4.1 Static analysis

Some tools rely on static analysis to identify bugs in error handling code. For example Rubio-Gonzalez et al [194] perform flow and context-sensitive static analysis to study error-code propagation in file systems. Their analysis tracks how error values propagate through the code and pinpoints places where they are overwritten, mishandled, or ignored.

Yuan et al [237] empirically analyze nearly 200 production failures in distributed systems and show that a significant fraction could have been prevented by three simple rules that their tool, Aspirator, checks for: empty exception handlers, overly generic or broad exception types in `try-catch` blocks, "`TODO`" or "`FIXME`" comments that accompany `try-catch` blocks.

In contrast, EPEx [111] detects error-handling bugs by checking developer-provided error specifications. The tool relies on symbolic execution to to ensure call sites satisfy these specifications along error paths. The key assumption is that software engineers can provide reasonably precise error specifications for widely used APIs.

Task cancellation is another recurring source of bugs in error handling and recovery mechanisms. Sethi et al [202] conduct an empirical study of over 200 task cancellation bugs across popular open-source systems. They find that failures originate both from the cancellation path itself as well as from the post-cancellation aftermath . They devise several anti-patterns based on these findings and design static checkers to identify these issues.

2.4.2 Dynamic analysis and testing

Other tools rely on dynamic analysis and software testing to trigger such defects. For instance, ExChain [139] performs exception dependency analysis using dynamic execution monitoring to reconstructs exception chains. It that analyzes the effects of program state changes related to exceptions. Based on this, it uses hybrid taint analysis to track those state propagation and establish potential causal links between exceptions.

EIO [92], in contrast, actively injects I/O faults to evaluate whether file systems correctly handle disk failures. Their paper shows that error-handling is only *occasionally* correct, as injected faults lead to silent data loss, crashes, or misbehavior.

2.5 Concurrency Bug Detection

2.5.1 Data race detection

Data races are among the most prevalent forms of concurrency bugs and occur when two threads access the same memory location concurrently without being properly synchronized, with at least one operation being a write while neither being synchronization primitives. A large body of work has focused on detecting such defects through both static and dynamic analysis.

Static race detectors analyze the program code to identify potential problematic memory accesses without executing the program. RacerX [69] uses a flow-sensitive, interprocedural lockset analysis to detect potential race conditions and deadlocks. LockSmith [193] similarly performs static analysis by combining lockset reasoning with pointer aliasing to verify whether shared memory variables are properly protected by locks. DCUAF [18] focuses on concurrency use-after-free bugs in Linux device drivers by combining pointer tracking, lockset reasoning, and control-flow analysis. It statistically identifies potentially function entry-points that can execute concurrently, then checks whether one may free a shared object while another accesses it unsafely.

Several works aim to improve the scalability and usability of static race detection. RaceD [25] proposes a compositional program analysis framework that analyzes each function independently and supports incremental analysis, allowing it to scale to large, evolving codebases. Echo [240] and D4 [151] integrate static race detection directly into the IDE, providing software engineers with immediate feedback as they write code. Both tools rely on a combination of static pointer and happens-before analysis to detect unsynchronized memory accesses.

Dynamic race detectors, in constrast, monitor execution to detect data races that actually manifest at run time. Eraser [196] was among the first to introduce lockset-based dynamic race detection by tracking which locks protect which memory locations and labeling inconsistent locksets as potential races. Goldilocks [68]improves detection precision by enforcing locking properties via instrumentation and guarantees soundness with respect to a defined “data race free” memory model.

ThreadSanitizer [201] uses a hybrid lockset and happens-before technique, maintaining metadata for each memory access while computing potential races based for dynamic interleaving. It has been widely adopted in practice at Google, including in Valgrind, LLVM, and the Go runtime. FastTrack [76] optimizes the traditional happens-before algorithm of ThreadSanitizer by replacing full vector clocks with compact epoch tracking for the common case where few threads compete on the same variable. This reduces instrumentation overhead at the expense of precise and race-finding capabilities.

KARD [5] is a lightweight dynamic data race detector for races caused by inconsistent lock usage. It leverages Intel MPK to dynamically enforce per-thread exclusive access to shared objects within critical sections, such that any conflicting access is caught via a protection fault. KARD combines key-enforced access, consolidated object-to-page mapping, automated object tracking, and fault-pruning techniques to maintain low runtime overhead while reliably detecting harmful data races.

2.5.2 Active delay injection

Active delay injection techniques aim to expose concurrency bugs by perturbing thread schedules at runtime in a targeted manner. These approaches typically begin with execution tracing or static analysis to identify suspicious memory access patterns or thread interactions, and then validate them by injecting delays at strategic program points to increase the likelihood of harmful interleavings.

TSVD [143] focuses on exposing thread-safety violations by monitoring calls to known thread-unsafe APIs and injecting short delays in one thread when two such calls are observed to overlap. The tool tracks runtime invocations to likely unsynchronized methods (e.g., those that misuse Java collections) and identify potentially problematic call pairs when two threads concurrently invoke the same unsynchronized operation. TSVD then immediately injects a carefully crafted delay into one of the threads to increase the likelihood of the two API invocations to overlap. This lightweight execution feedback driven strategy exposes latent thread-safety violations within one or two test executions, allowing the tool to uncover thousands of bugs in open-source and commercial software with zero false positives.

CTrigger [182] aims to trigger atomicity violations by identifying likely atomic code blocks using dynamic analysis, then deliberately injecting delays to perturbing thread scheduling around those program regions. The tool injects delays to delay one thread's entry into a block while letting another interfere, enabling it to expose hidden atomicity violations.

ConMem [246] targets memory consistency bugs by detecting memory operations whose side-effects are not correctly ordered across threads. It focuses on whether memory accesses produce externally visible access inconsistencies. The tool first observes candidate problematic access pairs, then validates them by enforcing their reordering by carefully pausing threads at strategic points in the execution.

DCatch [154] addresses distributed concurrency bugs, a class of defects that arise from incorrect operation interleavings in distributed processes. These bugs are triggered when messages between nodes attempt to access or modify shared distributed state in an unintended order. The tool first monitors the regular execution of the system to infer potential bug candidates by analyzing observed message-passing behavior against a set of predefined happens-before rules that model safe distributed interactions. DCatch next replays the execution while injecting targeted delays at RPC boundaries to deterministically enforce the suspected message orderings.

FCatch [156] targets a specific subclass of distributed defects, called time-of-fault bugs (ToF) which share similar characteristics with concurrency bugs. These occur when the effects of fault injection, such as node crashes or message delay/drop, interact with specific message timings and thread interleavings. In particular, a ToF bug manifests only if a fault is injected at a moment when the system is in a vulnerable state, like when a critical operation has begun on one node but has not yet propagated to others, resulting in inconsistent or incorrect behavior. FCatch detects and reproduces such defects by selectively delaying messages and injecting faults during vulnerable windows, exposing bugs that traditional fault-injection tools miss.

RaceFuzzer [198] exposes data races by first running an off-the-shelf static race detection stage to identify potentially problematic racy access pairs. Next, it uses a dynamic testing stage that instruments the target program and injects random delays before one or both instructions in each candidate pair, forcing them to execute in close temporal proximity. The last stage thus filters out false positives introduce by the static data race detector and validates true data races by observing actual faulty behavior at run time.

RaceMob [123] combines static analysis with crowdsourced bug validation. Like Race-

Fuzzer, it first uses a static data race detector to identify candidate memory accesses that might race. It then instruments the program at those locations with stubs that can randomly inject delays, and distributes the instrumented binaries to a network of end users for each to test a subset of them. RaceMob thus attempts to amortize the high cost of validating the likely data races identified through static analysis, by dispersing the work among multiple users.

Unlike the other tools discussed above, SherLock [142] aims to identify synchronization operations in a target application. It operates by first collecting regular execution traces during test runs and applying unsupervised clustering techniques to infer which instructions are logically related through implicit synchronization. It then re-executes the program with targeted delay injection at inferred conflicting instruction pairs to determine whether reordering cannot occur or leads to a crash, thus confirming the presence or lack of synchronization.

2.5.3 Systematic concurrency testing

Systematic concurrency testing [29, 77, 81, 82, 100, 133, 171] aims to explore alternative thread schedules within bounded execution spaces, by leveraging deterministic replay, scheduler instrumentation, model checking, or targeted heuristics to uncover interleaving-dependent bugs.

For example, CHESS [171] systematically explores thread schedules by controlling the operating system's thread scheduler to enumerate possible interleavings. It limits the state space by a predefined upper bound and uses deterministic replay to reproduce discover bugs.

In a followup work, PCT [29] introduces probabilistic concurrency testing techniques by using a randomized scheduler that guarantees probabilistic coverage of all bugs with a bounded number of thread preemptions. The key insight is that low-depth concurrency bugs can be exposed with high probability using a small number of strategic context switches, called scheduling constraints in the paper. For example, the authors show that triggering order violations requires only two such scheduling constraints.

SKI [77] systematically explores kernel thread interleavings without modifying the kernel itself. It operates by running the kernel in a custom virtual machine monitor that can pause and resume kernel threads under test, ensuring full control over the interleavings executed. Similarly, Snowboard [82] targets kernel concurrency bugs by exploring triggering inputs and thread interleavings using *potential memory communication* (PMC). PMC profiles sequential kernel tests to identify read/write memory conflicts, then generates and groups concurrent test pairs guided by PMC patterns to selectively schedule them with targeted interleavings through thread yields.

Snowboard [82] introduces a kernel concurrency testing framework for the Linux kernel that jointly explores test inputs and interleavings. It constructs *potential memory communication* (PMC) candidates by running tests sequentially and recording shared memory interactions, then schedules only those candidate test pairs concurrently using controlled scheduling to validate bugs.

2.5.4 General concurrency bug detection

Several works aim to detect broader classes of concurrency bugs such as more generic ordering and atomicity violations, improper syncrhonization use, and memory consistency errors, by analyzing program structure, memory behaviors, or symbolic interleavings.

UFO [104] uses predictive program analysis to detect concurrent use-after-free bugs without delay injection. It operates on execution traces and uses constraint solving to determine whether memory reuse following a free could race with a later access. Unlike techniques that actively inject delays, UFO aims to guarantee soundness without re-execution.

Portend [122] is a tool that differentiates between harmful and benign data races. It combines dynamic happens-before analysis with symbolic execution and execution replay to explore alternate orderings of racing events. It thus avoid false positives inured in traditional race detectors by attempting to force the problematic ordering and observe racing interleaving at run time.

ConSeq [245] generalizes the ideas behind ConMem (see above) by inferring concurrency bugs through flagging anomalies in sequential executions. It identifies thread interleavings that could lead to incorrect output values and perturbs the schedule at runtime through active delay injection to validate whether the reordering leads to a fault. This approach allows it to detect atomicity violations and memory ordering bugs in multithreaded applications.

ConVul [33] detects concurrency memory corruption vulnerabilities by first identifying pairs of memory-access instructions which might race and then enforcing alternate interleavings through a custom scheduler. When the reversed execution leads to observable crashes, the tool confirms the presence of a concurrency use-after-free or similar error, eliminating false positives by enforcing manifestation.

2.6 Performance Bug Detection

Researchers have developed numerous tools to detect performance bugs [4, 66, 117, 146, 174, 175, 177, 231, 232] by identifying inefficient code patterns, costly loops, expensive repeated computations, and suboptimal utilization of data structures.

Jin et al. [117] conducted an empirical study of 109 real-world performance bugs and developed rule-based detectors for inefficient code patterns, uncovering hundreds of previously unknown issues in MySQL, Apache, Mozilla, and other applications. Toddler [175] flags loops performing redundant work by spotting similar memory-access patterns across iterations, while Caramel [174] detects inefficient loops that can be optimized with non-intrusive fixes. Olivo et al. [177] checks for asymptotically suboptimal uses of collections (e.g., iterative traversals causing quadratic complexity) to combat memory-related bloat. Other profiling tools [66, 231, 232] identify excessive object creation and unused data, track runtime copying to find object bloat, trace inefficient memory allocations and pinpoint low-tenancy data structures that consume resources without proportional benefits.

More recently, PCatch [146] predicts cascading performance bugs by running the target application under small workloads that ship with the target application, then statically locating potentially non- scalable code (e.g., loops) and using this information to dynamically built an extended happens-before graph that model causal and resource contention events. It further tracks the origin of each to see whether a slowdown remains local or propagates. This hybrid analysis allows PCatch to forecast source-to-sink slowdowns before they occur at production scale.

2.7 Software Analysis and Monitoring Frameworks

2.7.1 Generic dynamic instrumentation frameworks

A variety of run-time analysis frameworks have been developed to identify a wide variety of bugs, including a diverse set of memory defects. AddressSanitizer [200] and Valgrind [172] are two well-known toolkits in this space. AddressSanitizer is a memory error detector that instruments code at compile time to find buffer overflows and use-after-free bugs during program execution. It uses a shadow memory and custom allocator to mark invalid regions, allowing it to detect errors at the exact point of occurrence with reasonable overhead (73% average slowdown for the original implementation [200]). Valgrind, in contrast, is a heavy-weight dynamic binary instrumentation framework for building comprehensive code analysis tools. For every register and memory location, Valgrind can maintain a shadow replica that tracks metadata such as whether a byte is defined or tainted. This design enables designing powerful checkers (e.g., its Memcheck sub-system for detecting uses of uninitialized memory [214]) at the cost of significant execution overhead, making Valgrind-based analyses slower but extremely thorough. Both AddressSanitizer and Valgrind have inspired numerous specialized bug-finders. For instance, several tools targeting use-after-free errors build on these instrumentation capabilities to monitor object lifetimes and detect incorrect reuse of freed memory [31, 88, 132, 160, 222].

2.7.2 Performance monitoring frameworks

Large-scale distributed systems require different monitoring and tracing techniques to diagnose and mitigate performance failures. Most notably, Microsoft's Magpie [20] and Google's Dapper [206] infrastructure record request-flow traces to identify operational slowdown, and Meta's Mystery Machine [43] reconstructs end-to-end request models from logs to identify which steps contribute the most to latency. Pivot tracing [164] introduced dynamic causal monitoring, allowing software engineers to inject queries that correlate events across systems at runtime to find the root causes of slowness. Similarly, Retro [163] is a framework designed for multi-tenant services that experience contention and "noisy neighbor" slowdowns. It works by introduces targeted resource management by continuously monitoring each tenant's resource usage across all components of a distributed system and exposing this information to a central policy engine that can enforce performance goals at different system control pointsflanagan09flanagan09. More recently, PerfSig [96] analyzes system metrics, logs, and method-level traces to isolate unusual slowdown patterns. It then applies information-theoretic causality analysis to generate cached ⟨`anomaly, root-cause function`⟩ signatures pairs automatically.

2.8 AI-guided software engineering

The rise of AI and in particular, LLMs in recent years, has brought opportunities to software engineering research, with an emergent set of papers applying LLMs to code generation [158, 181, 242], testing [53, 120, 136], repair [75, 110, 230], analysis [54, 144, 145, 184], summarization [6, 7], and documentation [213].

2.8.1 AI support for Test generation

An emerging line of research explore how AI can support and enhance test generation. For example, CodaMosa [136] augments search-based input fuzzing with LLMs to escape coverage plateaus. It runs an off-the-self fuzzer until coverage no longer increases, identifies call statements that rarely executes, prompts an LLM to generate inputs that attempt to cover them, and adds these to the fuzzer's input corpus.

Libro [120] use LLMs as "few-shot testers" by iteratively prompting them to generate tests based on bug specifications. It then relies on traditional filtering and triaging techniques to rank synthesized tests for usage. In contrast, Titan [53] use LLMs as "zero-shot fuzzers" by having them perform both generation-based and mutation-based fuzzing. It uses one LLM agent to generate the code, and another agent to perform small variations while provided with the LLM-generated input and an run-time execution profile.

Recently, Liu et al. [159] introduced EvalPerf which "synthesizes a (input) synthesizer" by prompting an LLM with chain-of-thought few-shot learning to produce a test input sampler controlled by a parameter "scale". EvalPerf uses exponential input sampling to generate challenging but computable inputs, mainly focusing on length stressing inputs or on generic, commonly known patterns.

2.8.2 AI-guided code generation and translation

Several works investigate how AI can assist with code generation and translation tasks. For example, Codeditor [242], a tool for a multi-language code evolution relines on a fine-tuned LLM to propagate changes to two code versions of a project. The tool models code changes as token edit sequences which the LLM can iteratively reason about the intermediary code differences and correlate those across the two target programming languages.

Pan et al [181] conduct a large-scale study on LLM-assisted code translation and conclude that LLMs correctly perform this task less than half of the time (2%–47%). Similarly, EvalPlus [158] focuses on evaluating LLM-generated code and evolve existing benchmarks in this space.

2.8.3 AI-assisted program repair and synthesis

LLMs have also been successfully applied to automate program repair, synthesis, and even formal proofs.

Xia et al [230] conducted an extensive study on automated program repair using 9 recent LLMs fine-tuned on code. They evaluated the models on standard bug-fix benchmarks, using prompts that either ask for full function rewrites, one-line patches, or code infills. The study found that off-the-shelf LLMs can fix substantially more bugs (12%–47%) than prior automated program repair tools, also outperforming the state-of-the-art learning-based repair techniques.

Baldur [75] uses LLMs to generate and repair formal proofs in a proof assistant (Isabelle [183]). It replaces traditional search-based step-by-step proof construction, with a model fine-tuned to produce an entire proof script in one shot. If the proof fails to verify, a secondary model suggests repairs by considering the failure and error message, similarly to patching code. The tool produces nearly 10% more more theorems than previous sophisticated, state-of-the-art auto-provers.

2.8.4 AI-informed program analysis

Several works explore how to effectively integrate AI with traditional program analysis.

Pei et al [184] investigates how LLMs can help inferring program invariants They train a code model to predict likely invariants (e.g., loop invariants, pre/post-conditions) from source code by fine-tuning it on a dataset of true invariants mined by tools like Daikon [71] The authors enable better model reasoning by prompting the LLM to first generate invariants at intermediate points in the code, mirroring abstract interpretation strategies, before actually synthesizing the invariant at the target location. In a followup paper, Ding et al [54] aims to address the semantic gap in model pre-training on code. The authors argue that conventional code models learn only from static text and thus struggle with tasks requiring understanding of run-time behavior. To solve this, they design TRACED, which provides the model with sample inputs and the corresponding runtime outputs/coverage of code during pre-training, so it internalizes dynamic properties. TRACED improves pre-trained code models by about 12% for execution path prediction and by approximately 25% in run-time variable value predictions

LLift [145] integrates LLMs with static analysis to triage use-before-initialization warnings in large codebases (e.g., the Linux kernel). It identifies undecided candidates by prompting an LLM to identify code context that either rules out infeasible paths or confirms feasible initialization data flow. The LLM's response is then provided back to the static analyzer module to prune out and re-prioritize the suspected use-before-init it identifies. On Linux, LLift analyzed approximately 300 suspected use-before-initializtation cases with 50% precision and no missed bugs, revealing 13 previously unknown bugs.

2.8.5 AI-assisted code summarization and documentation

LLMs are showing promising results for code summarization, documentation and specification mining tasks.

Ahmed et al [6, 7] demonstrate how LLM can effectively assist software engineers with code summarization. They first explored few-shot specialization for code summarizers [6] and showed that LLMs can be project-tuned with only a handful of examples from a target codebase to significantly improve the summaries for that particualr codebase (2%–46%, or 12% on average). They subsequently designed ASAP [7] to automatically augments prompts with semantic clues extracted from code (e.g., parameter roles, return values, simple control-flow facts). ASAP parses each function to mine these inferred semantic code properties and prepends them to the LLM input, yielding a 11–16% improvements over specialized code mining tools.

HotGPT [213] explores the use of LLMs making software documentation more directly useful for tasks like bug finding and performance tuning. The authors considered use cases for reading Javadoc comments to generate precise exception conditions, parsing configuration manuals to identify performance-related settings, and interpreting design documentation to extract correct lock usage rules. In each case, a general LLM model was prompted to translate free-form text documentation into a more formal or checkable format (e.g., a boolean predicate, a label of whether a config affects performance). HotCPT achieves accuracy on par with or better than prior specialized tools (e.g. LearnConf [141]) that were custom-built for each documentation type: 100% specificity and 54%–90% sensitivity when identifying locking-rule comments and 70%–86% with dedicated prompts); 81% accuracy (83% precision / 76% recall) on performance-config identification versus 76% accuracy (87% and 46%, respectively) for LearnConf.

2.8.6 Test and training benchmarks for AI-guided tools

A significant amount of research focuses on creating and evolving high-quality benchmarks that can evaluate AI's software engineering capabilities. For example, APPS [97], MBPP [15], HumanEval [38] and EvalPlus [158] work on curating high-quality Python benchmarks including coding problems along with correctness tests. HumanEval-X [249], MultiPLe [35], and MBXP [13] extend the Python tasks into other programming languages. Finally, PIE [205] and Effi-Learner [102, 103], two recently developed AI-guided code optimization tools, along with EvalPerf [159], released their synthesized tests as benchmarks, to help evaluating code efficiency.

CHAPTER 3
ANALYSIS AT THE STATEMENT LEVEL: SCHEDULE-GUIDED MEMORY ORDERING BUGS DETECTION

As a first step, this chapter explores how to efficiently apply the execution perturbation and test reutilization principles described in Chapter 1, and shows how active delay injection can help infidelity concurrency bugs that require code analysis at the lowest level of granularity, the *program statement level*.

This chapter fucuses on MemOrder bugs—a type of concurrency bug caused by incorrect timing between a memory access to a particular object and the object's initialization or deallocation. We first show experimentally that the current state-of-the-art delay injection technique leads to high overhead and low detection coverage since MemOrder bugs exhibit particular characteristics that cause high delay density and interference. Based on these insights, we propose Waffle, a delay injection tool that tailors key design points to better match the nature of MemOrder bugs. Evaluating our tool on 11 popular open-source multi-threaded C# applications shows that Waffle can expose more bugs with less overhead than state-of-the-art techniques.

This chapter is, in part, a reprint of the material that appears in Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys 2023) [211].

3.1 Motivation

Concurrency bugs are difficult to detect and time-consuming to diagnose. Typically, they only manifest under rare timing conditions [29, 121, 135, 143], many escaping rigorous in-house testing and causing severe production failures [79, 99, 109, 220].

Among the various approaches to detect concurrency bugs before code release, *active*

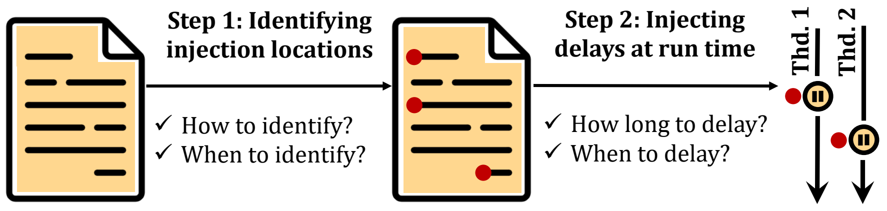


Figure 3.1: The workflow of active delay injection

delay injection [70, 123, 143, 182, 198] has an inherent advantage of high detection accuracy. Specifically, this approach reports a bug only *after* it manifests as a *consequence* of the delays injected. However, how to apply active delay injection to a wide variety of concurrency bugs, not only with high accuracy but also with low overhead and high bug coverage, is still an open question.

As Figure 3.1 illustrates, active delay injection identifies strategic program locations in a target program where concurrency bugs may exist (Step 1). Then, at run time, it injects delays at those particular locations attempting to trigger rare timing conditions and increase the chances of exposing concurrency bugs (Step 2). However, despite its inherent accuracy advantage, active delay injection traditionally suffers from high overhead and potentially low bug coverage.

When identifying injection locations (Step 1), there is a tension between the analysis cost and the quality of the locations identified. On the one hand, pruning program locations already synchronized and hence not needing delay injection, requires costly synchronization analysis [182, 198], such as happens-before analysis [130] (typically, 5–10× slowdowns reported by prior studies [76, 143]). On the other hand, performing no synchronization analysis results in too many injection points [70, 123] and hence high injection overhead.

When carrying out delay injection (Step 2), there is a trade-off between the cost per run and the number of runs needed to expose bugs. Prior work [70, 123] typically samples only a few injection locations in each run, lowering the overhead per run at the expense of needing many runs to expose bugs.

Recently, TSVD [143] effectively adapted active delay injection to detect a particular type of concurrency bug—thread-safety violations—not only with high accuracy but also with low overhead and high coverage. In Step 1, TSVD completely abandons the expensive happens-before analysis and instead relies on easy-to-measure physical time gaps between operations to infer which program locations are likely un-synchronized. In Step 2, TSVD injects delays aggressively in each run to minimize the number of runs needed to expose bugs.

Although effective, TSVD leaves behind an intriguing question: Does its unique approach to active delay injection work for broader classes of concurrency bugs? And if not, are there other design changes to active delay injection that work?

We investigate these questions by focusing on MEMORDER bugs—a type of concurrency bug caused by the lack of synchronization between access to a memory object and its initialization or deallocation (disposal).

It is crucial to expose MEMORDER bugs before software release as they are both common and severe. According to previous studies, MEMORDER bugs are the dominant type of order violations [162, 204]. Moreover, they lead to memory corruption which can cause crashes [88, 104] and serious security vulnerabilities [132, 227].

MEMORDER bugs also present significant research challenges as they have drastically different location and timing properties from thread-safety violations, well representing real-world concurrency bugs beyond those tackled by TSVD:

- *Location-wise*, thread-safety violations can only occur at call sites of thread-unsafe APIs [143]. In contrast, MEMORDER bugs can occur at any memory access to shared heap objects, which are much more common;

- *Timing-wise*, exposing a thread-safety violation requires the execution windows of two thread-unsafe API calls to overlap. Conversely, exposing a MEMORDER bug requires memory accesses to an object to occur before the object's initialization or after its deallocation/disposal (see Figure 3.2). These represent two fundamentally different

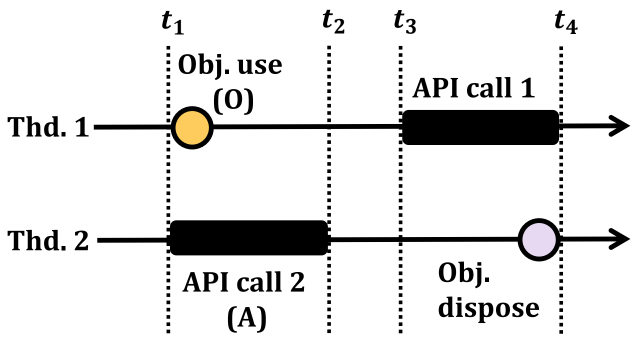


Figure 3.2: Different timing conditions. To make API calls 1 and 2 execute concurrently, the delay length needs to be within a range: $(T_4 - T_1)>$ delay$_A$ $> (T_3 - T_2)$. In contrast, to force an object use after it has been deallocated, the delay needs to be sufficiently long: delay$_O$$>(T_4 - T_1)$.

concurrency-bug timing conditions: atomicity violations for the former and order violations for the latter [162].

3.1.1 *Adapting the state-of-the-art*

We began our investigation by adapting the delay injection design of TSVD to detect MEMORDER bugs. Unfortunately, the efficacy of the resulting tool—referred to as WAFFLEBASIC—is limited. Our evaluation reveals that WAFFLEBASIC injects delays at a much higher rate than TSVD, and struggles with greater overhead and lower bug detection coverage.

This experience led us to decompose the workflow of active delay injection into four key design points as illustrated in Figure 3.1—to understand why TSVD's approach does *not* work for MEMORDER bugs and propose a new design that would.

How to identify delay candidate locations? Analyzing WAFFLEBASIC reveals that, due to their location properties, there are simply too many program locations where MEMORDER bugs can manifest for TSVD's inference heuristics to prune effectively. In turn, this leads to abundant (dense) and often pointless delay injection.

In particular, we observe that many delays injected by WAFFLEBASIC are needlessly

Design decisions	RaceFuzzer [198]	CTrigger [182]	RaceMob [123]	DataCollider [70]	TSVD [143]	WAFFLE
How to identify candidate location?						
Synchronization analysis?	✓	✓	✓	×	×	✓*
Synchronization inference?	×	×	×	×	✓	✓
When to identify candidate locations?						
During delay injection runs?	×	×	×	×	✓	×
How long is the delay?						
A fixed-length delay	✓	✓	×	✓	✓	×
When to inject at run time?						
Avoid delay interference?	n/a	n/a	n/a	n/a	×	✓
At sampled candidate loc.?	✓	✓	✓	✓	×	×
Probabilistic injection?	×	×	✓	✓	✓	✓

Table 3.1: Different design decisions for recent active delay injection tools. ("*" indicates partial analysis is done; "n/a" stands for «not applicable» as these tools deal with sparse sets of injection locations.)

attempting to reverse the execution order between memory accesses across the boundaries of thread forks. This particular type of synchronization can be analyzed accurately with negligible run-time overhead by using a special type of *thread-local storage*, a feature available in modern languages such as Java and C#. Then, the number of delays injected at run time can be reduced considerably with little analysis cost.

When to identify candidate locations? To minimize the number of bug-detection runs, TSVD uses the (buggy) location identification heuristic in the same run where it injects delays. Specifically, after identifying a program location ℓ where a bug may exist, TSVD considers injecting a delay the next time ℓ is executed during the same run.

We found such a strategy not suitable for WAFFLEBASIC. This approach results in abundant (dense) delay injection which severely affects the efficacy of location identification which relies on physical time information. Furthermore, such a strategy often cannot help expose MEMORDER bugs in one run, as many of their candidate locations like object initialization/disposal execute only a small number of times (often once) in each run.

Consequently, separating location identification and delay injection in different runs could fit MEMORDER bugs better.

How long is the delay? TSVD uses a fixed delay length for all candidate locations. Relying on this strategy combined with having to handle a denser set of candidate locations for MEMORDER bugs results in an unfortunate trade-off between longer delays required for bug-exposing capabilities and shorter delays for lower run-time overhead.

In comparison, injecting delays with different lengths at different locations is a more suitable strategy for MEMORDER bugs: the long delays required for exposing certain bugs will not lead to unnecessary delays at many other program locations.

When to inject delay at run time? To minimize the number of runs, TSVD injects delays at candidate locations with high probability[1] compared with previous work, while also allowing multiple threads to pause simultaneously. Unfortunately, due to the location property of MEMORDER bugs, this strategy leads to severe delay overlapping in WAFFLEBASIC, with many delays canceling each other. Such delay interference (and ensuing cancellation) occurs almost deterministically for some MEMORDER bugs, due to their unique timing properties.

Therefore, MEMORDER bugs require more careful coordination during delay injection.

3.1.2 A new design

Guided by these insights, we propose WAFFLE—a delay injection tool for exposing MEMORDER bugs that explores new trade-offs and heuristics for each design point discussed above. Given a program under test, WAFFLE runs it once to identify candidate injection locations with lightweight happens-before analysis. In subsequent runs, often just one, WAFFLE carries out delay injection with a carefully designed delay-or-not decision-making process guided by the information collected from the first run.

Table 3.1 summarizes WAFFLE's design decisions, and shows how our tool compares to TSVD and other recent delay injection techniques—techniques that rely on expensive program analysis and/or need many runs to expose bugs, as discussed above.

We implemented WAFFLE for C# and evaluated it on 11 popular open-source multithreaded C# applications. Our experiments show that WAFFLE successfully exposes 18 MEMORDER bugs, including 12 known and 6 *previously unknown* issues, without any bug-related prior knowledge. WAFFLE manages to reliably discover and trigger most of these bugs (15 out of 18) in just two runs. The end-to-end slowdown is only 2.5× compared with running the bug-triggering input once without any instrumentation. Our evaluation also

1. Deterministically injecting delays at every candidate location is widely considered unacceptable, due to the huge overhead and delay interference.

shows that WAFFLE exposes more bugs with much less overhead than various alternative designs.

Starting from an attempt to understand why TSVD works so effectively for thread-safety violations and whether a simple adaptation suffices for MEMORDER bugs, our study ends up with a completely different active delay injection design and a new tool, WAFFLE. We hope our journey sheds light on how to architect active delay injection tools for broader classes of concurrency bugs. Consequently, we make WAFFLE publicly available [2].

3.2 Background

As mentioned in §3.1, our first goal is to understand the state-of-the-art for active delay injection. Thus, we begin by presenting a few details about TSVD [143].

TSVD is an active delay injection tool that aims to expose thread-safety violations (TSVs), a type of concurrency bug that manifests when the execution windows of two thread-unsafe API calls operating on the same object overlap. Empirically, it reports significantly more TSVs in proprietary software with much less overhead than traditional delay injection techniques [143]. To achieve this, TSVD works as follows:

How to identify delay candidate locations? Given a program binary, TSVD first instruments program locations where thread-safety violations are likely to occur, namely call sites of thread-unsafe APIs [143]. At run time, TSVD collects information at these instrumentation points and leverages two heuristics to maintain a set $\mathcal{S}$ of thread-safety violation *candidates*. Each candidate consists of a pair of program locations $\{\ell_1, \ell_2\}$ where API calls at ℓ_1 and ℓ_2 may contribute to thread-safety violations. Consequently, ℓ_1 and ℓ_2 are candidate locations where TSVD injects delays and exposes potential thread-safety violations. We refer to $\mathcal{S}$ as the *candidate set*.

2. https://github.com/bastoica/waffle

The first heuristic—near-miss tracking—*adds* candidate pairs to $\mathcal{S}$ based on constraints related to the 2 corresponding threads, objects accessed, and the physical timestamps of the operations involved. Specifically, if one thread-unsafe API is invoked at location ℓ_1 from thread thd_1 accessing object obj_1 at time τ_1, while another is invoked at location ℓ_2 from thread thd_2 accessing obj_2 at time τ_2, TSVD adds $\{\ell_1, \ell_2\}$ to $\mathcal{S}$ iff. $obj_1 = obj_2$, $thd_1 \neq thd_2$ and $|\tau_1 - \tau_2| \leq \delta$, for a time gap threshold δ (called the *near-miss window* [143]). The intuition is that two thread-unsafe APIs accessing the same object from different threads are more likely to cause a thread-safety violation if they execute close to each other at run time.

The second heuristic—happens-before inferencing—*removes* candidate pairs from $\mathcal{S}$, namely those unlikely to trigger thread-safety violations, based on delay injection feedback. Assume TSVD added a candidate pair $\{\ell_1, \ell_2\}$ to $\mathcal{S}$. When a delay is injected before ℓ_1 in thread thd_1, TSVD checks whether it causes a proportional slowdown before location ℓ_2 in thread thd_2. If true, TSVD infers that there is a likely happens-before relationship between ℓ_1 and ℓ_2 and consequently removes $\{\ell_1, \ell_2\}$ from $\mathcal{S}$.

When to identify candidate locations? To minimize the number of bug-detection runs, TSVD uses the above two heuristics to dynamically update $\mathcal{S}$ during the same run it injects delays in. After adding $\{\ell_1, \ell_2\}$ into the candidate set, TSVD injects a delay before ℓ_1 at the immediate next opportunity, especially if ℓ_1 is exercised again in the same run.

How long is the delay? TSVD relies on fixed-length delays. Specifically, TSVD injects a `Thread.Sleep()` operation of δ milliseconds before a candidate location (e.g. ℓ_1). The configuration of δ balances the performance and bug-exposing capabilities: when δ is too short, many bugs are missed; when δ is too long, delay injection overhead becomes prohibitive.

When to inject at run time? For each candidate pair $\{\ell_1, \ell_2\}$, TSVD injects a delay with 100% probability when ℓ_1 is exercised for the first time. However, each time this action fails to expose a thread-safety violation, the probability of injecting a delay before ℓ_1 in the future drops by a small constant $\lambda > 0$. When this probability reaches 0, all candidate pairs involving ℓ_1 are removed from $\mathcal{S}$. This decay constant is carefully set: If λ is too small, many ineffective delays would contribute to an overhead increase. If too large, only a few candidate locations are delayed, thus many runs are needed to thoroughly search for bugs. We refer to this strategy as *probability decay*.

Additionally, TSVD injects delays aggressively and allows multiple threads to be blocked in parallel, to reduce the number of runs needed to expose thread-safety violations. Although delays injected simultaneously could overlap creating interference and thus canceling each other's effect, the sparsity of candidate locations related to thread-safety violations, combined with the probability decay scheme avoid such interference in most situations [143].

3.3 WAFFLEBASIC: Adapting TSVD

In our first attempt to detect MEMORDER bugs using active delay injection, we design WAFFLEBASIC by adapting TSVD [143] for this type of bug. On the one hand, WAFFLEBASIC departs from TSVD by operating on different program locations, relevant to MEMORDER bugs (§3.3.1). On the other hand, WAFFLEBASIC preserves TSVD's core delay injection philosophy (§3.3.2). Unfortunately, these design choices combined with the location and timing properties of MEMORDER bugs limit WAFFLEBASIC's efficacy (§3.3.3).

3.3.1 How to identify delay candidate locations?

Instrumentation sites WAFFLEBASIC first instruments every program location where a MEMORDER bug could occur. Specifically, WAFFLEBASIC instruments all operations related to reference-type variables, namely access to member fields and calls to member methods of heap objects. For each operation, WAFFLEBASIC records the corresponding object ID, physical timestamp, the operation type, and the active thread.

At run time, an instrumented operation is categorized into one of three types: object initialization, object disposal, and object use. An operation that changes the object's reference from `NULL` to non-`NULL` is considered an *object initialization.* An operation that changes the state of the object's reference from non-`NULL` to `NULL` or makes an explicit call to the object's destructor (i.e., `Dispose()` method) is considered an *object disposal.* Calling one of the object's member methods or accessing one of its member fields is considered an *object use.*

Adding to the candidate set $\mathcal{S}$. WaffleBasic adapts the near-miss heuristic of Tsvd to match the characteristics of MemOrder bugs. Consider an object initialization (object use) at location ℓ_1 executed by Thread 1 on object obj_1 at time τ_1, and another object use (object disposal) at location ℓ_2 executed by Thread 2 on object obj_2 at time τ_2. WaffleBasic adds $\{\ell_1, \ell_2\}$ to $\mathcal{S}$ as a candidate use-before-initialization (use-after-free) MemOrder bug iff. $obj_1 = obj_2$, $thd_1 \neq thd_2$, $\tau_2 - \tau_1 < \delta$, where δ is the size of the near-miss window.

Specifically, $\{\ell_1, \ell_2\} \in S$ forms a MemOrder *bug candidate* and ℓ_1 becomes a *candidate location* where WaffleBasic injects delays attempting to expose this potential (candidate) MemOrder bug. In other words, WaffleBasic injects delays before an object initialization hoping to force it to execute after its corresponding object use (as recorded in $\mathcal{S}$), and before an object use hoping to force it to execute after its corresponding object disposal (as recorded in $\mathcal{S}$).

Removing from the candidate set $\mathcal{S}$ WaffleBasic similarly adapts the happens-before inference heuristic of Tsvd. Given a candidate pair $\{\ell_1, \ell_2\}$, WaffleBasic checks whether a delay injected before ℓ_1 is observed to block the progress of the other thread right before ℓ_2. If the delay propagates, ℓ_1 and ℓ_2 are likely ordered by a happens-before relationship and WaffleBasic removes the pair from $\mathcal{S}$.

3.3.2 What about the other design decisions?

The remaining design decisions of WAFFLEBASIC follow those of TSVD.

When to identify candidate locations? WAFFLEBASIC injects delays in the same run in which it adds or removes pairs from the candidate set $\mathcal{S}$. This way, WAFFLEBASIC attempts to expose MEMORDER bugs in a minimal number of runs.

How long is the delay? Similar to TSVD, WAFFLEBASIC injects delays of a fixed length δ. This constant is set to 100 milliseconds, exactly as in TSVD.

When to inject delays at run time? Like TSVD, WAFFLEBASIC implements a *probability decay* scheme. Similarly, WAFFLEBASIC injects delays at candidate locations in $\mathcal{S}$ with a probability that starts at 100% and gradually decreases towards 0 if no MEMORDER bugs could be uncovered there. Likewise, WAFFLEBASIC also allows multiple delays to block multiple threads in parallel.

3.3.3 How effective is WAFFLEBASIC?

We evaluated WAFFLEBASIC using 11 open-source applications with 12 previously reported MEMORDER bugs (details in Tables 3.3 and 3.4). Our experiments found that although WAFFLEBASIC can expose some MEMORDER bugs, it fails to expose others even after many runs. Experiments also reveal that WAFFLEBASIC incurs a significant overhead for several applications. In particular, we observed several shortcomings of WAFFLEBASIC that reflect the fundamental difference between MEMORDER bugs and thread-safety violations. These shortcomings led us to the design of WAFFLE (§3.4).

Too many program locations in play. MEMORDER bugs and thread-safety violations have different location properties. Thus, WAFFLEBASIC needs to handle a much more nu-

App	Instrumentation Sites		Injection Sites	
	TSV	MO	TSV	MO
ApplicationIns.	8.7	188.6	0.1	3.5
FluentAssert.	57.3	76.9	0.3	5.9
Kubernetes	5.6	338.5	1.5	3.8
MQTT.Net	23.2	544.1	7.9	156.6
NetMQ	49.2	619.0	13.5	143.4
NSubstitute	1.3	261.4	0.6	10.7
NSwag	2.2	110.4	0.3	70.8
Ssh.Net	56.3	179.0	0.4	13.1

Table 3.2: The average number of unique static instrumentation and delay-injection sites for thread-safety violations (TSV) and MemOrder bugs (MO) across all test inputs.

merous set of instrumentation sites than Tsvd (i.e., program locations that access heap objects versus thread-unsafe API call sites). For 8 out of 11 applications[3] in our benchmark suite, WaffleBasic's instrumentation sites are over 10× more common than Tsvd's, in most cases (see "Instrumentation Sites" columns in Table 3.2). With a larger base to start with, more program locations tend to pass the near-miss heuristic at run time and get added to the candidate set $\mathcal{S}$. Similarly, the number of delay injection locations identified by WaffleBasic is predominantly an order of magnitude more than those identified by Tsvd (see "Injection Sites" columns in Table 3.2).

Too much delay overlap. A denser set of instrumentation sites means more delays injected at run time which, in turn, increases delay overlap. To quantify this overlap we run every test suite for the benchmarks in Table 3.2 and compute the complement of the ratio between the "time projection" of all delays over the total delay value injected. This way, if no delays overlap, the value is 0 while if all delays overlap the ratio is close to 1 (i.e., $\frac{D-1}{D}$, with D representing the total number of delays injected at run time). For Tsvd, the average overlap is less than 1% for 6 out of the 8 applications, with the remaining 2 applications showing an average overlap of 12% and 15%, respectively. In contrast, for WaffleBasic,

3. The public version of Tsvd cannot instrument the other 3 applications in our benchmark suite.

the average overlap is $2 - 28\%$, with 3 applications above 25%.

Too few dynamic instances. The main reason TSVD detects many thread-safety violations in just one run is that most thread-unsafe API calls are executed multiple times per run, offering multiple chances for bug manifestation [143]. Unfortunately, this is not true for MEMORDER bugs. In particular, many objection initialization operations naturally execute a small number of times each run. In our evaluation, the median number of dynamic instances for all object initialization operations is 2 across all unit tests for all applications.

3.4 WAFFLE: A New Design

To tackle the challenges faced by WAFFLEBASIC, we propose WAFFLE. As illustrated in Figure 3.3, WAFFLE works as follows:

- First, WAFFLE executes the targeted program binary once, without injecting any delay, to record a delay-free execution trace. We refer to this as the *preparation run.*
- Next, WAFFLE analyzes this unperturbed trace to (i) construct the set of candidate location pairs $\mathcal{S}$, (ii) determine the delay length for each candidate location, and (iii) identify which candidate locations might interfere with each other (§3.4.1—3.4.4).
- Finally, WAFFLE carries out delay injection in subsequent runs using information collected during the preparation run as well as from the ongoing run itself (§3.4.4). We refer to these as *detection runs.*

In the rest of this section, we describe WAFFLE's design decisions and how they differ from WAFFLEBASIC.

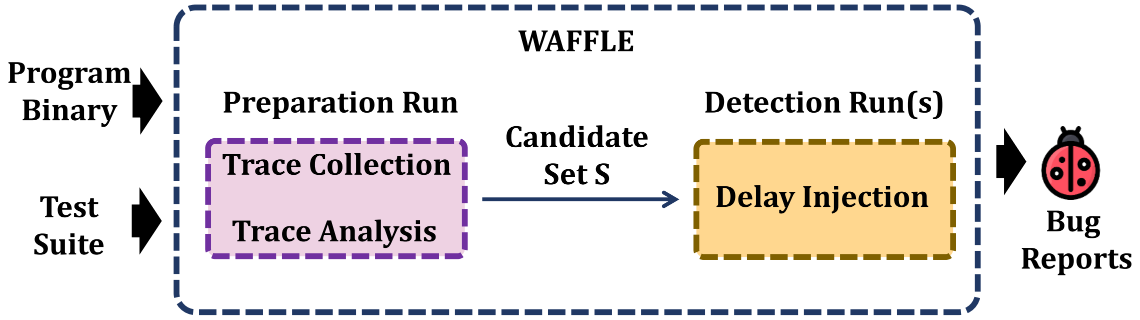


Figure 3.3: Workflow diagram of WAFFLE. Our tool operates in two stages. First, it collects execution feedback by running the original test suite, uninstrumented and injection free. Second, it uses the information gathered in the first stage to guide delay injection in subsequent instrumented, “delay active” runs. With a handful of exception, WAFFLE needs only one “delay active” run (so two, in total) to expose MEMORDER bugs.

3.4.1 How to identify delay candidate locations?

What went wrong in WAFFLEBASIC? To identify candidate locations, WAFFLEBASIC (like TSVD) completely disregards traditional happens-before analysis. Instead, it uses run-time heuristics (near-miss tracking and happens-before inference) to *infer* what operations may be un-synchronized and hence should be part of the candidate set $\mathcal{S}$. Unfortunately, this design did not work well for WAFFLEBASIC, affecting detection overhead and bug coverage, for several reasons.

First, MEMORDER bugs naturally present many more instrumentation sites than thread-safety violations (Table 3.2) which contributes to an increased number of delays getting injected and higher run-time overhead.

Second, the happens-before inference can be less accurate in WAFFLEBASIC due to delay overlaps. WAFFLEBASIC (like TSVD) infers happens-before relationships between two locations ℓ_1 and ℓ_2, iff. delaying Thread 1 before exercising ℓ_1 causes a proportional slowdown of Thread 2 before exercising ℓ_2. However, if a second delay is injected around the same time in Thread 2, thus overlapping with the first in Thread 1 (see diagram in Figure 3.5), the happens-before inference heuristic cannot reliably determine whether the slowdown in

Thread 2 is caused by a synchronization operation or it is solely the effect of the second delay. Consequently, the more delays overlapping, the less the effective happens-before inference heuristic is.

Finally, even when the happens-before inference is as accurate as in TSVD, it offers little help to those locations that only execute a small number of times each run (e.g. object initializations). By the time WAFFLEBASIC infers the happens-before relationship, the program locations involved no longer get exercised during that same run.

Design of WAFFLE At first glance, WAFFLE may need to go back to full-blown happens-before analysis, which would require significant manual effort in annotating synchronization operations, in addition to the high overhead incurred by the happens-before analysis itself [142, 143].

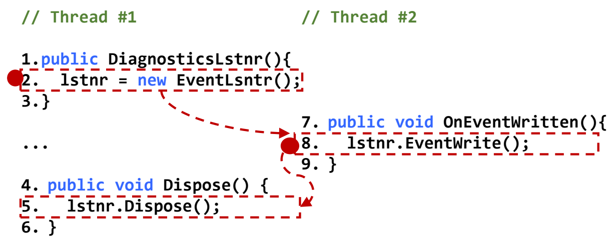
Fortunately, we found that a sizable fraction of MEMORDER bug candidates is causally ordered by a specific type of happens-before relationship — that between parent and child threads. Typically, this happens because many objects are allocated in a parent thread before the worker threads are created. Consequently, WAFFLE replaces the happens-before inference in WAFFLEBASIC with a parent-child thread relationship analysis. Pruning these ordered MEMORDER candidates during the preparation run reduces the number of candidate program locations where WAFFLE has to inject delays. As Table 3.7 shows, failing to remove them decreases the average performance of our tool by 1.17×. However, the impact on memory-intensive applications is much greater (e.g. 1.73× for NpgSQL)

To track parent-child thread relationships, traditionally we would need to instrument every program location where the parent forks a child thread. This is challenging in modern object-oriented languages such as Java and C# which have multiple mechanisms for thread creation. Instead of instrumenting various types of thread fork operations, WAFFLE leverages a special type of thread-local storage (TLS) that automatically gets copied from a parent to all child threads at the moment of thread creation. This language feature is supported by other modern object-oriented programming languages such as Java (`InheritableThreadLocal` [219]) and C++ (`LogicalCallContext` [218]).

Note that while WAFFLE only considers threads, .NET provides a similar mechanism for task-oriented programming — *async-local storge* — which supports state propagation from a parent to a child *task* irrespective of which thread these tasks are scheduled to run on.

WAFFLE tracks happens-before relationships induced by thread forks by implementing vector clocks on top of the TLS mechanism. More precisely, WAFFLE creates and stores a tailored thread-local vector clock object in the TLS memory region of each thread. This vector clock is represented by a set of tuples $\{(tid_1, \&rctr_1), (tid_2, \&rctr_2), ...\}$, with each tuple representing a thread ID and a reference (pointer) to the corresponding logical time counter. When a child thread is created, the TLS memory region of the parent thread (and, consequently, the vector clock object with it) gets automatically propagated to the child thread. At this point in the execution, WAFFLE allocates a vector clock for the child thread using information from the vector clock "cloned" from its parent. Specifically, WAFFLE implements the constructor of the vector clock object to (1) append a tuple $(tid_k, \&rctr_k = 1)$, with tid_k being the child thread ID, to the vector clock content copied from the parent thread; and (2) increment the logical counter of the parent using the counter reference (pointer) passed through the TLS. Note that while vector clock updates happen as part of the TLS propagation which is triggered by a thread fork operation, the parent's vector clock remains inaccurate until TLS region is completely copied to the child thread—as the value is not incremented right before the fork happens. However, WAFFLE makes no vector clock comparisons in this time frame.

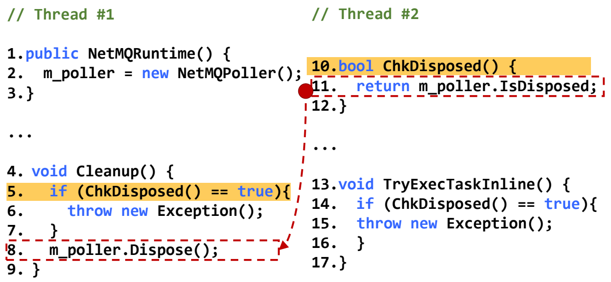
WAFFLE leverages these vector clocks during near-miss tracking. Consider a candidate location pair $\{\ell_1, \ell_2\}$ that satisfy all requirements related to timing, active threads, and memory access set forth by the near-miss tracking heuristic (§3.4.1). WAFFLE additionally checks whether the corresponding vector clocks of the two active threads at time τ_1 and τ_2,



```
// Thread #1                          // Thread #2

public DiagnosticsLstnr(){
  lstnr = new EventLsntr();
}
                                      public void OnEventWritten(){
...                                     lstnr.EventWrite();
                                      }
public void Dispose() {
  lstnr.Dispose();
}
```

(a) ApplicationInsights Issue #1106 [1]: Interfering bugs



```
// Thread #1                          // Thread #2

public NetMQRuntime() {
  m_poller = new NetMQPoller();       bool ChkDisposed() {
}                                       return m_poller.IsDisposed;
                                      }
...
                                      ...
void Cleanup() {
  if (ChkDisposed() == true){         void TryExecTaskInline() {
    throw new Exception();              if (ChkDisposed() == true){
  }                                     throw new Exception();
  m_poller.Dispose();                   }
}                                     }
```

(b) NetMQ Issue #814 [2]: Interfering candidate locations

Figure 3.4: Examples of delay interference

respectively, cannot be partially ordered before deciding to add the pair to $\mathcal{S}$ or not.

3.4.2 When to identify candidate locations?

What went wrong in WaffleBasic? The design decision of combining candidate locations identification and delay injection into the same run does not benefit detecting MemOrder bugs as much as it benefits detecting thread-safety violations. This happens because program locations involved in many MemOrder bugs have only a few dynamic instances, as mentioned in §3.3.3. Consequently, starting delay injection in the same run makes little difference for program locations that execute a few times, if at all, after being identified as candidate locations.

Furthermore, our evaluation shows that the injected delays sometimes interfere with candidate location identification, which relies on physical time information. For example, a pair of candidate locations $\{\ell_1, \ell_2\}$ observed during a delay-free run may "disappear" once delays are injected, as delays injected between ℓ_1 and ℓ_2 could prevent them from executing close to each other, failing the $|\tau_1\text{-}\tau_2| < \delta$ requirement. Intuitively, the more delays injected at run time, the more severe this interference is.

Design of Waffle In contrast, Waffle conducts a preparation run (delay-free) for planning purposes (see Figure 3.3), before injecting delays in subsequent, detection runs. In this first run, WAFFLE uses the near-miss heuristic together with the parent-child relationship pruning to construct the candidate locations set $\mathcal{S}$. In subsequent runs, WAFFLE injects delays at these locations. In addition, WAFFLE leverages the delay-free environment in the first run to collect timing information to help guide delay injection. We will elaborate more in the next two sub-sections.

Note that using a delay-free run can potentially increase the cost of WAFFLE, as at least two runs are now needed to expose a MEMORDER bug. We believe the benefits outweigh the extra cost, and we experimentally validate this claim during evaluation (§3.6).

3.4.3 How long is the delay?

What went wrong in WAFFLEBASIC? WAFFLEBASIC struggles to find a delay length that can balance performance and bug-exposing capabilities. Compared to TSVD, WAFFLE-BASIC incurs significantly more overhead under the same delay length setting due to starting with a larger candidate set $\mathcal{S}$ (§3.3.3). For example, using a delay length of 100 milliseconds, TSVD incurs 15%, 9%, and 11% overhead when running all multi-threaded tests available for ApplicationInsights, FluentAssertion, and Kubernets.Net, respectively [143]. In contrast, WAFFLEBASIC incurs over 100% overhead for the same three applications (Table 3.5).

Of course, we could lower the overhead by using a much shorter delay. However, the bug-exposing capabilities of WAFFLEBASIC would suffer. For example, decreasing the delay length from 100 to 10 milliseconds would speed up the average performance of WAFFLEBASIC by about 4 times across all multi-threaded unit tests available for NetMQ. Unfortunately, in that case, the known MEMORDER bug [2] which could be exposed by WAFFLEBASIC when utilizing delays of 100 milliseconds, cannot be triggered with delays of only 10 milliseconds even after many runs (§3.6.2-§3.6.3).

Design of Waffle To address this challenge, Waffle leverages the observation that different bugs have different time gaps between corresponding operations in bug-free runs (i.e., the gap between an object initialization and its use, or between an object use and its disposal) and hence opts to inject delays of different lengths at different locations. In particular, for the 12 known bugs in our evaluation (§3.6.1), measurements reveal that these time gaps range from less than 1 to around 100 milliseconds. Consequently, if we observe the time gap between ℓ_1 and ℓ_2 to be much shorter than ℓ_3 and ℓ_4 during a delay-free run, we can inject much shorter delays at ℓ_1 than at ℓ_3 during detection (i.e., delay injection) runs.

To achieve this, Waffle keeps track of time gaps between each pair of candidate locations in $\mathcal{S}$. For a candidate pair $\{\ell_1, \ell_2\} \in S$, Waffle creates a record with the delay length at that particular candidate location, $len_{\ell_1} = |\tau_1 - \tau_2|$. If ℓ_1 is part of multiple candidate location pairs (e.g. $\{\ell_1, \ell^*\} \in S$), Waffle updates len_{ℓ_1} with the larger gap (i.e., $len_{\ell_1} = \texttt{MAX}(|\tau_1 - \tau_2|, |\tau_1 - \tau^*|)$.

During a detection run, Waffle injects delays proportional to the gap measured above. For instance, given a location ℓ_1, Waffle injects a delay of $\alpha \cdot len_{\ell_1}$ milliseconds, for a small constant $\alpha \geq 1$. In our experiments, $\alpha = 1.15$.

3.4.4 When to inject at run time?

What went wrong in WaffleBasic? As discussed in §3.3.3, WaffleBasic experiences much more delay overlap than Tsvd. These overlapping delays interfere with each other and cause WaffleBasic to miss some MemOrder bugs with significant probability. Most often, this occurs in two scenarios:

1. Interfering bugs: Sometimes, the manifestation of two bug candidates interfere with each other — one requires Thread 1 to execute faster than Thread 2, and the other requires Thread 2 to execute faster than Thread 1. When attempting to trigger them both, the injected delays cancel each other. Unfortunately, these cases are particularly common when, for example, attempting to trigger a use-before-initialization and use-after-free MemOrder bug on the same object instance (in the same run).

Figure 3.4a illustrates such an example from ApplicationInsights [1]. The bug manifests when the constructor takes longer than expected to allocate `lstnr` before a `WRITE` event occurs, which triggers the `OnEventWritten()` handler. WaffleBasic consistently misses this bug because it injects delays both before the allocation at line 2 in Thread 1, aiming to push the allocation after the object use (line 8), and before the object use at line 8 in Thread 2, aiming to push the use after the dispose operation (line 8). Therefore, WaffleBasic blocks both threads in parallel for the same duration and cannot trigger the bug even after 50 runs.

2. Interfering dynamic instances: Sometimes, WaffleBasic injects a delay at a location ℓ_1, hoping to make it execute after ℓ_2. Unfortunately, this delay repeatedly gets canceled out by a delay at another dynamic instance of ℓ_1, which is executed by the same thread right before exercising ℓ_2.

Figure 3.4b illustrates such an example from NetMQ [2]. The failure happens when a connection is abruptly terminated, causing several shared objects to be disposed (e.g. `m_poller` on line 8, Thread 1) while other threads are still processing network packages (e.g. `m_poller` on line 11, Thread 2). To trigger this bug, WaffleBasic injects a delay right before line 11, aiming to push the use of `m_poller` in Thread 2 to execute after the dispose operation in Thread 1. Unfortunately, since line 11 is also executed (in a different execution context) by Thread 1 right before the dispose call, both threads get delayed at around the same time and for the same duration. This, in turn, prevents WaffleBasic from exposing the bug even after 50 runs.

Design of Waffle Similar to WaffleBasic and Tsvd, Waffle uses the *probability decay* strategy to inject delays at candidate locations with decreasing probabilities. Unfortunately, this alone is not enough to mitigate delay interference. Thus, WAFFLE proposes an additional, new heuristic to reduce delay interference.

A naïve solution to the delay interference problem is to modify WAFFLEBASIC to inject only one delay per run, similar to prior work [182, 198]. However, this would require too many runs to expose the bug, as WAFFLE routinely observes tens, or even hundreds of location candidates after the preparation run, for just one input. A better solution might be to change WAFFLEBASIC to avoid any parallel (i.e., overlapping) delays. However, completely avoiding parallel delays may cause some bugs to take many runs to trigger [143]. This could be even worse for MEMORDER bugs: If the candidate location ℓ of a bug executes only a small number of times each run, WAFFLEBASIC may never trigger the bug if another candidate location, from a different thread, keeps getting exercised right before ℓ does.

Therefore, WAFFLE proposes a new heuristic that aims to strike a balance between enabling parallel delay injection and reducing delay overlaps. The high-level idea is to enhance WAFFLEBASIC so that a delay planned to be injected before ℓ_1 is skipped when an *interfering* delay is ongoing. As illustrated in Figure 3.5, we consider a delay planned for ℓ^* in thread Thd$_2$ to interfere with another delay planned for ℓ_1 in a different thread Thd$_1$ if two conditions are met: (1) First, ℓ^* executes before ℓ_2 on the same thread Thd$_2$—if $\{\ell_1, \ell_2\}$ is a bug candidate that WAFFLE aims to expose by injecting a delay before ℓ_1, allowing another delay before ℓ^* will block Thd$_2$, essentially canceling out the original delay and preventing ℓ_1 execute after ℓ_2. (2) Second, ℓ^* needs to either execute shortly ahead of ℓ_1 or between ℓ_1 and ℓ_2—otherwise, the interference is negligible.

A challenge in implementing the above strategy is that we cannot predict whether ℓ_2 will be executed by thread Thd_2 (i.e., the thread of ℓ^*) at the time when execution reaches ℓ_1. Consequently, we cannot predict which delays might interfere with ℓ_1 when it gets exercised. To address this challenge, we leverage the preparation (i.e., delay-free) run. Specifically, we augment the near-miss heuristic as follows: when the execution reaches ℓ_2 and WAFFLE

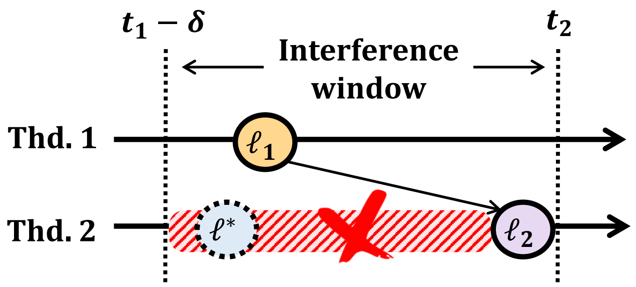


Figure 3.5: Illustration of delay interference. Highlighted, the interference window — when a concurrent delay injected in Thread 2 can cancel the effect of the delay injected before ℓ_1.

identifies $\{\ell_1, \ell_2\}$ as a candidate pair, it further checks if any other candidate location, say ℓ^*, was exercised by the same thread that reaches ℓ_2 at a time between $t_1 - \delta$ or and t_2. If so, the pair (ℓ_1, ℓ^*) is added to a global set $\mathcal{I}$ of locations that could interfere with each other if delayed simultaneously. $\mathcal{I}$ (along with $\mathcal{S}$ and the per-location delay-length values) is saved after analyzing the execution traces recorded during the preparation run and used to bootstrap future detection runs. This way, in future detection runs no delay gets injected at ℓ^* as long as there is another delay concurrently injected at a location interfering with ℓ^* (e.g., ℓ_1). Conversely, no delay gets injected at ℓ_1 as long as there is another delay concurrently injected at a location interfering with ℓ_1 (e.g. ℓ^*).

3.5 Implementation

WAFFLE We implemented WAFFLE to find MEMORDER bugs in C# applications. WAFFLE measures 6,378 lines of C# code and an additional 433 lines of Python and PowerShell scripts. WAFFLE is divided into three key components: (1) the instrumenter which statically instruments the target binary; (2) the trace analyzer which constructs the candidate set $\mathcal{S}$, the interference set $\mathcal{I}$, and determines appropriate delay lengths; and (3) the runtime which implements the delay injection algorithm.

1. WAFFLE's instrumenter. This component takes an application binary as input and wraps every access to object member fields or calls to member methods in a proxy function.

The proxy function transfers control to WAFFLE's runtime library (see below) which implements the delay injection strategy. To instrument the binary, WAFFLE relies on a .NET instrumentation framework called Mono.Cecil [72].

WAFFLE executes the instrumented application at least twice. WAFFLE runs the instrumented application once to collect an unperturbed execution trace containing every access to heap objects (preparation). Note that no delay is injected in this preparation phase. WAFFLE runs the instrumented application at least one more time, to inject delays and expose MEMORDER bugs (detection).

2. WAFFLE's trace analyzer. This component analyzes a run time trace to identify pairs of memory accesses likely to cause MEMORDER bugs. At this stage WAFFLE constructs the candidate set $\mathcal{S}$, discarding those pairs ordered by happens-before relationships between parent and child threads, along with those with a physical time gap larger than the near-miss threshold. Next, it computes the appropriate delay length to inject for each candidate pair in $\mathcal{S}$. Finally, it identifies which candidate locations interfere with each other if delays are to be injected concurrently, constructing set $\mathcal{I}$.

3. WAFFLE's runtime. This component implements the core delay injection algorithm.

Recall that during the preparation run, the library logs all accesses to reference-type variables (heap objects) along with metadata such as timestamps, accessed object id, and access types (i.e., object initialization, object dispose, access to object fields, or call to object methods).

During the detection run, the library injects delays according to the delay planning done in the preparation run and updates delay probabilities at candidate locations based on the probability decay heuristic. After each detection run, the new delay probabilities are saved on disk and used to bootstrap the next detection run.

Finally, WAFFLE reports a bug only when the target binary raises a `NULL` reference exception as a consequence of the delay injection performed. At that time, the relevant runtime context (i.e., faulty input, candidate locations involved, stack traces for all threads, and delay value information) is recorded as part of the bug report.

Extending Waffle. To extend Waffle for applications written in other object-oriented languages, like Java and C++, we mainly need to change the underlying instrumentation framework. We would similarly instrument method calls and field accesses related to heap objects, and implement the same delay injection algorithms which rely primarily on language-independent time-based heuristics. One language-dependent feature used by Waffle is C#'s thread-local storage (TLS) mechanism which transfers state from a parent to a child thread and facilitates happens-before analysis (§3.4.1). However, similar language features are available for Java [219] and C++ [218].

WaffleBasic. Like Waffle, WaffleBasic uses the same .NET instrumentation framework (i.e., Mono.Cecil [72]). WaffleBasic instruments applications at similar locations as Waffle, although different delay injection algorithms are conducted at run time. Furthermore, WaffleBasic does not instrument applications to produce traces and does not contain a trace analyzer. The delay injection policy implemented by WaffleBasic measures 447 lines of C# code.

3.6 Evaluation

Benchmarks We evaluate Waffle on 11 popular open-source multi-threaded C# applications (Table 3.3). We made this selection by searching Github for C# applications that (1) are popular, measured by the number of Github stars; (2) have well-maintained test suites which would help us conduct a systematic evaluation; and (3) contain confirmed and clearly described MemOrder bugs in their issue trackers.

To find MemOrder bugs, we first searched for issue reports containing keywords such as "data race" or "race condition". We further narrowed down the results using keywords like

Application	LoC	# Multi-thread tests	# Stars
ApplicationInsights	151.2K	156	0.5K
FluentAssertions	47.7K	41	2.5K
Kubernetes.Net	173.2K	21	0.7K
LiteDB	18.3K	7	6.2K
MQTT.Net	27.1K	126	2.2K
NetMQ	20.7K	101	2.3K
NpqSQL	51.9K	283	2.4K
NSubstitute	17.9K	13	1.7K
NSwag	101.5K	18	4.9K
SignalR	51.8K	52	8.5K
SSH.Net	84.4K	117	2.8K

Table 3.3: Details about the set of open-source C# applications used to evaluate Waffle.

"exception" or "crash". Finally, we manually inspected the remaining reports to confirm they are describing MemOrder bugs and include bug-triggering inputs.

3.6.1 Methodology

Following instructions in these MemOrder bug reports, we were able to manually reproduce 12 previously known MemOrder bugs in 9 applications (the top 12 bugs in Table 3.3). These helped us evaluate the bug-detection capabilities of Waffle and WaffleBasic. Although we were unable to reproduce the MemOrder bugs reported in SignalR and MQTT.Net (we suspect the reported bug-triggering inputs or bug code versions are inaccurate), we keep these two applications in our benchmark suite as they both contain a large set of multi-threaded test cases.

Note that, although we manually reproduced all 12 known bugs, we did **not** use this knowledge when evaluating Waffle or WaffleBasic. Specifically, we ran both tools using *every* multi-threaded test case in the test suites of each application and recorded the number of bugs exposed as well as how many runs it took to trigger them, statistics about delays injected, and overhead measurements.

Experiments Setup We run each benchmark on a Windows 10 desktop machine with an Intel Core i7-8700 3.2GHz CPU, 16GB of RAM, and 1TB of disk space.

We use a near-miss window δ of 100 milliseconds for both Waffle and WaffleBasic—the default setting in Tsvd [143]. Similar to Tsvd, we also use 100 milliseconds as WaffleBasic's fixed-length delay value.

Finally, we repeat each experiment 15 times, to reduce measurement variations that could arise due to the probabilistic nature of our tools.

3.6.2 Bug-detection coverage

Waffle can trigger all 12 previously known MemOrder bugs, as well as 6 previously *unknown (i.e., unreported)* MemOrder bugs (18 bugs in total), using only inputs from the applications' test suites.

Note that none of these 18 bugs can manifest themselves without delay injection, even when we execute the corresponding bug-triggering inputs repeatedly 50 times. Moreover, some of the previously unknown bugs discovered by Waffle remained undetected for many months or even years (e.g. Bug-14). Additionally, Waffle can trigger 3 of the known bugs using a test case that was already available in the test suite before the issues were reported, indicating that our tool is useful for finding hard-to-detect bugs in mature software.

In contrast, WaffleBasic exposes only 11 out of the 18 bugs. WaffleBasic cannot expose any of the other 7 bugs even after a significant number of detection runs (50 in our evaluation). This happens because WaffleBasic injects many more delays and allows much more delay interference than Waffle, as discussed in §3.4.

In theory, the number of runs required to expose a MemOrder bug could vary in different attempts due to the probabilistic nature of concurrency bugs. Therefore, we repeated our experiment 15 times. When we report that a bug can be detected in 1 or 2 runs, we make sure that this is the case in the majority of attempts (i.e., at least 10 out of the 15 attempts).

No	Application	Issue ID	Previously known?	Exec. time (ms) w/o instrumentation	# of detection runs		Detection slowdown (×)	
					WaffleBasic	Waffle	WaffleBasic	Waffle
Bug-1	SSH.Net	80	Yes	2,464	2	2	1.4×	1.2×
Bug-2	SSH.Net	453	Yes	1,042	2	2	1.7×	1.6×
Bug-3	NSubstitute	205	Yes	437	1	2	3.3×	5.1×
Bug-4	NSubstitute	573	Yes	316	2	2	9.0×	4.4×
Bug-5	NSwag	3015	Yes	887	2	2	2.1×	1.8×
Bug-6	FluentAssertions	664	Yes	782	1	2	1.4×	2.7×
Bug-7	FluentAssertions	862	Yes	831	2	2	1.2×	2.5×
Bug-8	LiteDB	1028	Yes	495	-	2	-	4.9×
Bug-9	Kubernetes.Net	360	Yes	1,955	1	2	1.3×	2.0×
Bug-10	ApplicationInsights	1106	Yes	143	-	2	-	4.9×
Bug-11	NetMQ	814	Yes	18,503	5	2	5.1×	2.2×
Bug-12	NpqSQL	3247	Yes	1,097	-	4	-	6.9×
Bug-13	SignalR	n/a	No	952	-	2	-	1.3×
Bug-14	ApplicationInsights	2261	No	1,349	2	2	1.5×	1.3×
Bug-15	NetMQ	975	No	593	-	3	-	12.2×
Bug-16	MQTT.Net	1187	No	1,207	-	4	-	5.4×
Bug-17	MQTT.Net	1188	No	13,722	-	3	-	6.2×
Bug-18	Kubernetes.Net	n/a	No	1,494	2	2	2.5×	2.0×

Table 3.4: Detection results from WAFFLE and WAFFLEBASIC (Basic). WAFFLE discovered 6 previously unreported bugs (the bottom 6). Four of these bugs manifest in the latest available major release version (March 30th, 2022) and are reported by us. The other two (Bug-13, 18) no longer surface in the latest builds. The slowdowns are based on the execution time of the bug-triggering input without any instrumentation. "-" indicates that WAFFLEBASIC fails to expose the bug in 50 runs.

Bugs that require more runs to expose tend to behave more non-deterministically. For those bugs, we report the median number of runs required to expose them (Table 3.4).

Previously unknown bugs Among the 6 previously unknown MemOrder bugs exposed by Waffle, 4 have since been fixed by software engineers in the most recent releases.

Two out of these six bugs lead to use-before-initialization problems (Bug-13 and Bug-14). For example, Bug-14 in ApplicationInsights [1] happens because the constructor only manages to initialize the event handler field of the object (i.e., `this.buffer.OnFull`) before the "buffer event" occurs. At that point control is transferred to the event handler, which attempts to access one of the still uninitialized fields of the object, triggering a `NULL` reference exception.

The other four previously unknown bugs lead to use-after-free problems. For example, Bug-15 in NetMQ [3] happens because the underlying message queue gets disposed while the queue still stores messages that are currently being processed. Later on, when a worker thread attempts to dequeue a message that just finished processing, it triggers a `NULL` reference exception.

3.6.3 Bug-detection efficiency

For 14 out of the 18 bugs, Waffle reliably exposes them by running the corresponding test case twice. Specifically, the bug is reliably exposed in Waffle's first detection run after a preparation run. The remaining 4 bugs took Waffle 3 or 4 runs to trigger. This happens because NpqSQL, MQTT.Net, and NetMQ perform significantly more heap object accesses which, in turn, are the source of many more delay candidate locations for Waffle to sift through. Moreover, as shown in Table 3.4, Waffle incurs a 1.2×–5.1× slowdown (median, 2.1×) for these 14 bugs when compared with running the bug-triggering input without instrumentation. For 7 of those, the overhead is 2.0× or lower. This happens because the bug manifestation halts the detection run prematurely, thus the end-to-end time in these cases is similar to or much shorter than running the original test input twice without instrumentation.

The remaining 4 bugs take a longer time to get exposed (5.4×–12.2×), as they require more than one detection run to manifest. This happens because more delays get injected in each run due to the denser, much more frequent heap object accesses—a similar trend across all test inputs for these particular applications (Table 3.5). Note, however, that traditional race detection techniques routinely incur several times the slowdown (e.g. 5-10× [76, 143]).

In comparison, WAFFLEBASIC takes the same number of runs or, in one case, more (Bug-11), while managing to expose only 11 bugs. This is actually surprising, as WAFFLEBASIC starts delay injection from the first run, unlike WAFFLE which spends its first run for preparation, without injecting any delays. WAFFLEBASIC was able to expose only 3 bugs (Bug-3, Bug-6, and Bug-9) in fewer runs than WAFFLE (i.e., in its first run). WAFFLEBASIC requires more detection runs than WAFFLE for the remaining 8. Moreover, WAFFLEBASIC incurs longer end-to-end bug-detection overhead than WAFFLE for 7 out of these 11 bugs. Overall, these findings justify our decision to dedicate WAFFLE's first run for preparation without any delay injection (§3.4.2).

Finally, Bug-7 is the only case where WAFFLE incurs more overhead than WAFFLEBASIC, although both tools need the same number of detection runs to trigger it. This happens because WAFFLEBASIC exposes Bug-7 close to the beginning of its second run, while WAFFLE only does so towards the end of its second run (i.e., its first detection run).

3.6.4 Detailed results

Overhead. Table 3.5 reports the average overhead that WAFFLE incurs on *every* multi-threaded test case in each application's test suite, for both its preparation and detection runs. We exclude LiteDB since it contains only a few multi-threaded test cases, as noted in

App.	Base	WaffleBasic (%)		Waffle (%)	
	(ms)	Run#1	Run#2	R#1	R#2
Applic.	227	122	357	19	38
Fluent.	776	48	48	24	27
Kubernet.	2051	14	37	9	41
MQTT.Net	1768	TimeOut	TimeOut	13	332
NetMQ	1657	167	375	34	288
NpgSQL	1118	2818	2509	266	968
NSubst.	344	72	294	26	78
NSwag	995	12	56	14	51
SignalR	267	58	144	13	81
Ssh.Net	702	68	96	16	20

Table 3.5: Average overhead on all test inputs. (Base: the average run time of a test input without any instrumentation)

Table 3.3.

Waffle incurs much less overhead than WaffleBasic for 8 applications, and similar overhead for the remaining 2 applications (Kubernetes.Net and NSwag). Particularly, for NSubstitute, NpgSQL, and ApplicationInsights, Waffle's detection runs (R#2) are more than twice as fast as WaffleBasic's. Furthermore, for MQTT.Net, a protocol communication application, WaffleBasic incurs so much overhead that most of the test cases timed out, which does not happen for Waffle.

The performance benefit of Waffle comes from its decision to analyze parent-thread causal relationships (§3.4.1) and its reliance on variable-length delays (§3.4.3). This is reflected by the significantly fewer delays injected and the much shorter cumulative (total) delay duration introduced by Waffle, as illustrated in Table 3.6.

For 7 out of 10 applications (i.e., except for Kubernets.Net, NetMQ, and NSwag), Waffle injects only one-tenth to about half the number of delays across all test inputs, compared to WaffleBasic. Since Waffle uses variable-length delays instead of a fixed 100-millisecond value like WaffleBasic, the cumulative delay duration Waffle injects is 5× less than WaffleBasic. Note that for multi-threaded programs the cumulative delay du-

Application	WaffleBasic		Waffle	
	# Delays	Duration (ms)	# Delays	Duration (ms)
Applic.	2,475	247,500	475	7,212
Fluent.	448	44,800	43	167
Kubernet.	177	17,700	197	5,904
MQTT.Net	TimeOut	TimeOut	3,243	141,699
NetMQ	11,767	1,176,700	11,271	520,037
NpqSQL	246,477	24,647,700	123,166	4,535,586
NSubst.	575	57,500	78	1,083
NSwag	343	34,300	349	11,806
SignalR	861	86,100	513	11,342
Ssh.Net	829	82,900	506	10,126

Table 3.6: Cumulative number and duration of delays injected across all test inputs (one detection run for each input). TimeOut: most tests timed out due to excessive delay.

ration only indirectly affects the total execution time. Thus, although WAFFLE injects less cumulative delay than WAFFLEBASIC for Kubernets.Net and NSwag, the overhead incurred by WAFFLE and WAFFLEBASIC is similar. In all other cases, WAFFLE incurs much less overhead than WAFFLEBASIC.

Overall, WAFFLE achieves reasonable performance for in-house testing. For the preparation run (R#1), WAFFLE incurs 9–34% average overhead across all applications except for NpgSQL; for the first detection run (R#2), WAFFLE incurs 20–81% average overhead for all applications except for NpgSQL, NetMQ, and MQTT.Net. These three applications allocate a large number of objects at run time. Even though WAFFLE achieves significant improvements over WAFFLEBASIC, the delay density is still moderately high for these 3 applications.

Benefit of every design point Table 3.7 shows how different design points help WAFFLE's bug detection capabilities and performance. We measure the impact on the number of bugs exposed and overhead incurred, averaged across all test inputs for all applications when disregarding one of the four key design decisions discussed in §3.4 (e.g. w/o parent-child

	# bugs missed	slowdown over WAFFLE
no parent-child analysis (§3.4.1)	0	1.17x
no preparation run (§3.4.2)	4	1.84x
no custom delay length (§3.4.3)	1	1.03x
no interference control (§3.4.4)	6	1.41x

Table 3.7: Alternative designs detect fewer bugs with slower detection runs. (Baseline # of bugs and performance are from WAFFLE across all applications).

analysis means WAFFLE does not prune out parent-child thread causal relationships).

This experiment shows that every design point has its benefit. Among the 4, the decision of having a dedicated preparation run without delay injection (§3.4.2) and the decision of coordinating delays to avoid interference (§3.4.4) offer the biggest benefit in both bug coverage and performance. The other two are also helpful. For example, excluding parent-child causal analysis for NpgSQL slows down WAFFLE's detection runs by 1.73×, on average, across all test inputs.

False positives WAFFLE has no false positives, as our tool only reports a bug after triggering it *and* once it observes the NULL pointer exception not handled by the target application.

False negatives Although WAFFLE successfully detected all 12 previously known bugs as well as 6 previously unknown bugs in our benchmarks, it could still miss MEMORDER bugs for several reasons. First, like all dynamic detection tools, WAFFLE's bug detection capabilities rely on test inputs. If a buggy code region is not exercised by the test set, WAFFLE cannot detect the bug. Second, similar to TSVD, WAFFLE uses several algorithms that rely on physical time information, such as delay interference analysis, delay length analysis, near-miss tracking, and so on. Consequently, WAFFLE could non-deterministically miss some MEMORDER bugs in the first few detection runs.

3.7 Discussion

3.7.1 Limitations

Our evaluation shows the Waffle's potential for detecting a broader class of concurrency bugs. Yet, the current prototype has a few limitations.

First, Waffle's efficiency is ultimately linked to the number of heap memory accesses performed by the target application. More memory accesses mean more potential candidate-locations, thus more delays get injected which, in turn, translates to more slowdown, as discussed in 3.6.3.

Second, Waffle's delay interference detection algorithm is neither sound, nor complete. Waffle relies on a greedy approach, namely skip injecting a delay at location ℓ if an overlapping delay at another location ℓ' is ongoing. This strategy is not designed to capture *all* sources of interference (e.g. accumulated delays) and could potentially increase the number of runs needed to expose bugs if some critical delays are withheld for the first few injection opportunities. However, we did not observe the latter scenario in our evaluation and it is likely a rare occurrence as delays are injected probabilistically due to the probability decay heuristic.

Finally, Waffle relies on C# runtime support to identify parent-child thread relationships at low cost, by essentially inferring synchronizations determined by thread fork operations without having to explicitly monitor fork operations. To the best of our knowledge, no equivalent inexpensive runtime support exists for thread join operations. This creates an imbalance where more candidate-locations pertaining to object allocations get pruned during the (less expensive) analysis run, while more candidate-locations pertaining to object uses get pruned in the (more expensive) delay injection run(s).

3.7.2 Threats to Validity

Internal threats to validity. As described in §3.6.1, WAFFLE leverages existing tests suites shipped with the target application to expose MEMORDER bugs. Thus, our tool could potentially miss bugs that are not exercised by these inputs. Additionally, not all tests provided by software engineers exercise multi-threaded code, further narrowing bug coverage. Finally, WAFFLE incurs false negatives related to sources of delay interference and happens-before inferencing as discussed above.

External threats to validity. The 11 applications and 18 bugs in our evaluation (§3.6) may not be representative of real-world applications. Overall, we made a best-effort attempt to select non-trivial open-source C# applications that are both popular and broadly used (Table 3.3).

3.7.3 Relationship with prior work

Several techniques for concurrency bug detection aim to predict problematic interleavings by analyzing memory and synchronization traces from a single execution. As discussed in §2.5.1, many of these tools rely on an initial predictive or static analysis pass to identify candidate issues and then validate each through repeated delay-injection executions. Some of these tools do not aim to report bugs with no false positives since they do not attempt to actually trigger the bug. (e.g., [76, 173, 191, 196, 201]).

In contrast, tools discussed in §2.5.2 and §2.5.4 aim to trigger the bug and thus guarantee no false positives, yet they are not optimized for a small number of bug-finding trials and repetitions (e.g., [33, 104, 122, 154, 198, 245, 246]). These also incur about 10× or even 100× slowdown during the predictive bug detection run and require precise knowledge about synchronization operations present in the target application.

In contrast, WAFFLE detects MEMORDER bugs using only a small number of runs and typically avoids requiring more than two per-bug re-executions with a moderate overhead of

2.5× over the uninstrumented test runs.

Tools described in §2.5.3 systematically explore thread interleavings through exhaustive or bounded schedule manipulation, guided by metrics like synchronization coverage or probabilistic guarantees. These methods often incur substantial overhead or require costly instrumentation to effectively control scheduling. Conversely, Waffle applies lightweight binary instrumentation to run tests and uses execution feedback to steer them toward buggy schedules with minimal re-executions.

Input generation techniques, like those detailed in §2.3.1 geared towards correctness bugs, aim to synthesize inputs or method call sequences to surface latent concurrency bugs. Unlike these tools, Waffle repurposes existing unit and integration tests to uncover memory-order bugs without requiring new input synthesis.

Waffle targets nondeterministic concurrency bugs such as memory-order violations, which differ from deterministic memory safety issues like use-after-free or uninitialized reads. Tools covered in §2.7.1 effectively handle such intra-thread memory violations but are not designed to detect concurrency errors unless those bugs manifest spontaneously during stress testing. Waffle directly addresses this challenge by actively triggering rare interleavings during controlled executions.

Finally, past research has explored how to automatically infer happens-before causality between send/receive messages for system performance [4, 42], network dependency analysis [39], or concurrency bug detection [142]. These tools need to observe a large number of runs to make robust inferences and hence cannot directly help Waffle to expose bugs after a small number of runs.

3.8 Summary

This chapter explored a new design point in the active delay injection space. It demonstrated that active delay injection can be re-architected to expose a challenging class of concurrency bugs, MemOrder bugs, effectively and efficiently. Starting from an exiting state-of-the-art active delay injection tool, Tsvd, that focuses on thread-safety violations, we deconstructed the entire active delay injection workflow into four key design points: where to delay, when to decide, how long to pause, and how to coordinate multiple delays. We showed why simply transplanting design choices made by Tsvd (or similar tools) to MemOrder bugs yields higher overhead, low coverage, or both. We identified three obstacles that proved challenging for prior tools: (i) a dense explosion of injection candidate sites, (ii) interference and canceling effects between overlapping delays, and (iii) one-shot initialization/disposal events that rarely recur in the same run.

Guided by these insights, we proposed a new active delay injection design and built Waffle, a two-phase workflow that first records an unperturbed execution to perform lightweight happens-before inference and thread-local pruning, then injects variable-length, interference-aware delays in one or two subsequent detection runs to surface MemOrder bugs. Waffle repurposes existing tests that deployed with the target application, transforming them into bug-riggering tests, without burdening software engineers to write new ones.

While language-agnostic, Waffle focuses on C# software systems and exposed 18 MemOrder bugs (6 previously unknown) across 11 widely used open-source applications. It did so requiring fewer re-runs and with lower overhead than the state-of-the-art.

Most importantly, however, we believe the journey from Tsvd to Waffle sheds new light on this topic and can serve as a systematic blueprint for building resource-conscious active delay injection tools. We hope future work will extend these principles to other classes of (concurrency) bugs, integrate them with existing software testing tools, or even combine them with adaptive techniques that learn from program executions (e.g., AI).

CHAPTER 4
ANALYSIS AT THE MECHANISM LEVEL: SEMANTIC FAULT INJECTION FOR TRIGGERING BUGS IN RECOVERY LOGIC

This chapter investigates how to efficiently apply the two principles described at the beginning of this thesis by performing bug-finding at the *mechanism level*. Specifically, it explores how AI-guided semantic fault injection can help isolate defects in retry logic.

Retry—the re-execution of a task on failure—is a common mechanism to enable resilient software systems. Yet, despite its commonality and long history, retry remains difficult to implement and test.

Guided by a study of real-world retry issues [211], we propose a novel suite of static and dynamic techniques to detect retry problems in software. We find that the ad-hoc nature of retry implementation poses challenges for traditional program analysis but can be well suited for large language models; and that carefully repurposing existing unit tests can, along with fault injection, expose various types of retry problems.

This chapter is, in part, a reprint of the material that appears in the proceedings of the 30th Symposium on Operating Systems Principles (Austin, TX, USA) (SOSP 2024) [211]. This chapter focuses on the active testing of software systems through LLM-informed fault injection to uncover retry issues. It only offers a high-level overview of other static analysis that Wasabi leverages to increase its bug cover capabilities. The interest reader can find those additional design details, insights, and quantitive analysis in the full conference manuscript [211].

4.1 Motivation

Retry is a commonly used mechanism to improve the resilience of software systems. It is well understood that many task errors encountered by a software system are transient, and that re-executing the task with minimal or no modifications will succeed. However, retry can also cause serious or even catastrophic problems. Retry is oftentimes the last line of defense against various software bugs, hardware faults, and configuration problems at run time. Unfortunately, like other fault-tolerance mechanisms [40, 92, 111, 243], retry functionality is commonly under-tested and thus prone to problems slipping into production. Indeed, recent studies have identified a substantial portion of cloud incidents related to broken or unsafe fault-handling mechanisms, including that of retry [91, 105, 134, 155].

Despite its seeming simplicity, it is challenging to implement retry correctly. First, there are policy-level challenges regarding whether a task error is worth retrying and when to retry it. Often it is unclear which errors are transient and hence recoverable, and such retry-or-not policies require maintenance as applications evolve. It is also difficult to get the timing of retry correct: a system that retries too quickly or too frequently might overwhelm resources, while one that retries too slowly could lead to unacceptable delays in processing. Second, there are also mechanism-level challenges: how systems should perform retry—how to track job status, how to clean up the program state after an incomplete task, and how to launch a job again (and again)—continues to be prone to defects. These requirements are made more challenging by the fact that retry is not always a "simple loop": forms of retry that utilize asynchronous task re-enqueing, or circular workflow steps, whose implementation may be complex and difficult to identify, are common.

In recent years, a number of "resilience frameworks" or "fault tolerance libraries" have been developed to improve the resiliency of distributed applications, a major component of which has been configurable support for retry [64, 106]. But such frameworks, while helpful in some ways, cannot solve all policy or mechanism problems. While they support configuration of policy aspects (such as providing automated retry-on-error), they provide no help in deciding the policies, e.g. which errors should be retried; nor can they prevent issues in how retry is implemented. Moreover, their design can only support simple retry implementations. Instead, non-loop retry modes and retrying complex tasks—which are common—are difficult to support.

Testing retry logic presents similar challenges. To ensure reliability prior to deployment, software engineers typically run applications in a controlled, small-scale testing environment. However, recreating retry conditions requires software engineers to first, faithfully simulate *transient* errors that typically occur in production, and second, write specialized tests that exercise retry code paths with high-enough coverage and specially designed test oracles. Both are challenging and do not exist in today's unit testing frameworks.

The goal of this paper is to systematically study and characterize real-world retry bugs; and provide a solution to help improve this pervasive and critically important functionality in software systems.

Understanding the retry challenge. By thoroughly studying 70 retry-related incident reports from 8 popular open-source applications in Java [211], we find that the root causes of retry-related incidents are about equally common regarding (1) IF to retry a task upon an error (36%), (2) WHEN and how many times a task is retried (33%), and (3) HOW to properly retry without leaking resources or corrupting application states (31%).

By inspecting the retry code fragments in these incidents, we observe a broad diversity in how retry mechanisms are implemented, making it difficult to automatically identify them. There is no dedicated retry API in any of the cases we studied. In about 55% of the cases, the retry functionality is implemented as a simple loop, while in 45% of cases it is implemented as a non-loop structure, either as a finite state machine or using asynchronous task re-enqueing. Instead, we find comments, log messages, variable names, and error codes to offer the clearest evidence of a retry code structure.

By running and analyzing all unit tests of these 8 applications (thousands to tens of thousands for each application), we confirm that existing unit tests are poor at exposing retry bugs. Based on our analysis, close to 10% of the unit tests invoke a part of the application that could trigger retry. However, retry almost never occurs during unit testing as transient errors are extremely rare. About 0.1%–0.5% of unit tests in these applications contain a mechanism to deterministically inject transient errors, but they only test a tiny portion of retry logic (e.g., most of them check whether the injected exception can be caught or not) and are not capable of catching those most common retry bugs discussed above.

The interested reader can find an in-depth study, insights, and a quantitative analysis of identify retry in modern software systems in our full conference paper [211]. This chapter focus almost excursively on how to adapt the *active testing* principle that this dissertation relies on and designing an effective fault injection pipeline for detecting bugs in retry functionality.

Tackling retry bugs. Guided by these findings, we take a first step in enhancing the reliability of retry logic by developing WASABI [225], a toolkit combining static program analysis, large language models, fault injection, and unit testing to tackle all three types of retry-related bugs (IF, WHEN, and HOW problems mentioned above) in both loop and non-loop related retry. This suite of techniques uniquely enables bug-finding at the software mechanism level (i.e. retry), for which traditional program analysis is a poor fit.

WASABI operates in two workflows—a *dynamic testing* workflow and a *static checking* workflow—that complement each other. In the dynamic testing workflow, WASABI automatically alters the execution of unit tests using fault injection to exercise retry logic and expose retry bugs. WASABI does not require software engineers to create specialized tests, bug oracles, or other bug-finding policies. Instead, it uses large language models (GPT-4) and traditional static analysis to identify the locations of retry and the trigger exceptions of retry in the source code. This enhances traditional program analysis with the *fuzzy code comprehension* capabilities of large language models, allowing for more nuanced detection of retry logic. It then formulates and executes a fault injection plan, leveraging existing unit tests to probe these locations under simulated faults. Finally, WASABI comes with a set of test oracles that are specially designed to identify the manifestation of retry bugs.

In the static checking workflow, WASABI employs a combination of static control flow checks (CodeQL) and large language models (GPT-4) to identify retry-related bugs directly from source code. This approach extends the bug-finding capabilities of unit testing by allowing WASABI to check retry code not covered by existing unit tests. On the other hand, it can incur more false positives than unit testing and miss bugs that are related to system run-time states.

In all, WASABI identifies more than 100 distinct, previously unknown retry bugs in eight Java applications across all three types of retry-bug root causes. In particular, WASABI identifies 42 retry bugs by repurposing existing unit testing, and 87 through static analysis, with 20 bugs detected by both. Detailed comparison shows that repurposed unit testing, traditional static code analysis, and large language models each have their own limitations and can well complement each other in the task of detecting retry related bugs.

Finally, we make WASABI openly available to the broader community [1].

4.2 Wasabi

Given the different characteristics of IF-, WHEN-, and HOW-retry bugs, as discussed above, we have designed (1) a testing workflow that repurposes existing unit tests to expose WHEN and HOW retry bugs (§4.2.1) and (2) a set of static analysis routines and prompts that detect IF and WHEN retry bugs (§4.2.2). We refer to them together as WASABI [225].

4.2.1 Exposing retry bugs through unit tests

Modern systems all have extensive unit tests, typically thousands or more, carefully designed by software engineers. These unit tests offer a good opportunity to systematically exercise retry logic and expose bugs of various root causes and symptoms.

1. https://github.com/bastoica/wasabi

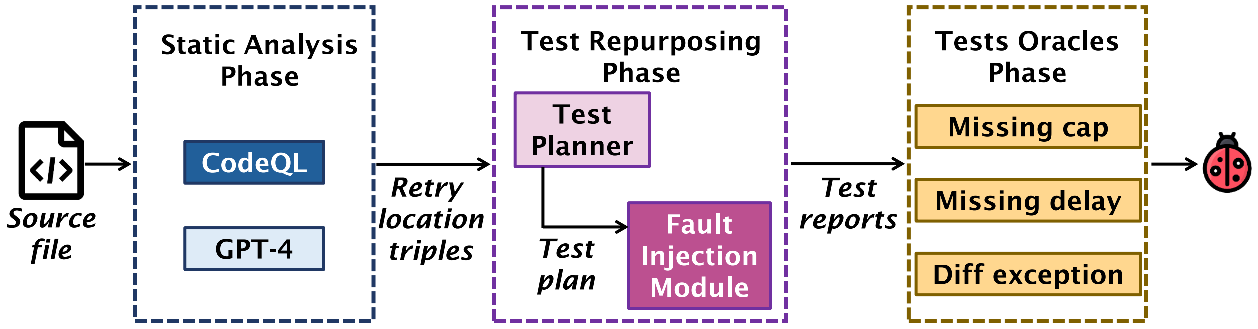


Figure 4.1: WASABI's repurposed unit testing workflow operates in three stages. First, our tool processes the codebase source file by source file to identify where retry logic is implemented by leveraging both control-flow analysis (CodeQL) and LLMs; additionally, it extracts exceptions that are likely to trigger those retry constructs. Second, it pairs tests with retry locations (test planner) in order to inject specific faults while exercise them by running that particular test (fault injection module). Third, the application behavior under fault injection is analyzed by three carefully crafted oracles to flag retry bugs.

However, directly running existing unit tests would not work for two key reasons. First, the triggers of retry are typically low-probability events like networking failures, and hence rarely occur during in-house testing. Second, even if a retry is triggered and a problematic retry logic is exercised, the problem may not be caught by existing test oracles that were designed without retry in mind. With these challenges in mind, WASABI uses the following components to repurpose existing unit tests and expose retry bugs:

1. A static analysis routine that processes program source code and reports where retry may happen (§4.2.1.1);
2. A fault injection module that simulates an exception to trigger retry when exercising unit tests (§4.2.1.2);
3. A set of test oracles specially designed to catch retry-related bugs (§4.2.1.3);
4. A test planner that coordinates among all unit tests so that all retry locations in unit tests will be exercised through fault injection in a cost-efficient way (§4.2.1.4).

```
class WebHdfsFileSystem {
  private HttpResponse run() throws IOException {
    for(int retry=0; retry < maxAttempts; retry++) {
      try {
          HttpURLConnection conn = connect(url);
          HttpResponse response = getResponse(conn);
          return response;
      } catch (AccessControlException e) {
          break;
      // AccessControlException may be wrapped. Fix
      + } catch (HadoopException he) {
      +   if (he.getCause() instanceof AccessControlException)
      +       break;
      } catch (ConnectException ce) {
      }
      Thread.sleep(1000);
    }
    return null;
  }

  private HttpUrlConnection connect(URL url) throws AccessControlException, ConnectException;

  private getResponse(HttpURLConnection conn) throws IOException;
}
```

Listing 4.1: Wrong Retry Policy: Non-recoverable error is retried (bug report HADOOP-16683, *Loop-based mechanism*). The `run()` method attempts up to `maxAttempts` to open an HTTP connection and read its response, returning immediately on success. If an `AccessControlException` or one wrapped in `HadoopException` is encountered (e.g., thrown by `connect()`), execution immediately breaks out of the loop without further retries. A `ConnectException` is explicitly caught and ignored, causing the `run()` method to sleep for one second and then retry. Any other `IOException` from `getResponse()` will propagate immediately, and if all retries exhaust without success, the method returns `NULL`.

For ease of discussion, we refer to a function M whose execution error causes itself to be re-executed as a *retried method*, such as `connect()` and `getResponse()` in Listing 4.1. We refer to the caller function of M that catches the error and conducts the retry as the *coordinator method* (e.g., method `run()` in Listing 4.1). The execution error of the retried method typically is reported to the coordinator through exceptions (70% of the cases in our study) or error codes (30%). The current prototype of WASABI focuses on exceptions, which we refer to as *retry triggers*. Finally, we refer to the call site of the retried method inside the coordinator method as a *retry location*.

4.2.1.1 Identifying retry locations

The main challenge is that retry logic does not have a unique code pattern for traditional program analysis to search for. For example, our extended work [211] reveals loop-based retry all involve try-catch blocks in loops, but there are also many loops with try-catch that do not offer retry—a method could be repeatedly invoked in a loop for processing different inputs, instead of for retry. Non-loop retry that involves asynchronous processing (e.g. task re-enqueuing) and state are even harder—it is difficult to tell whether a method would be re-executed from this code, not to mention whether the re-execution is for retry or not, and comprise 45% of the retry logic in our recent study [211] (Section 2).

To successful identify as much retry functionality as possible automaticaly, we propose a two-prong approach combining structural and non-structural cues—variable names, comments, and flow reachability—to surface retry-location triplets $\langle C, M, E\rangle$.

First, we use CodeQL [63] to single out any loop whose header is reachable from an in-body catch and whose body contains any string literals, variables, or methods whose names include "retry" or "retries" Specifically, we search for every call site C encapsulated in method M and throwing exception E whose handler jumps back to the beginning of the loop. Since, based on our study [211] (Section 2), nearly half of retry logic is implemented as non-loops structures, we rely on large language models—specifically, GPT-4—promoting themodel with an individual source file and asking to isolate retry functionality by reminding GPT-4 of the various shapes this functionality can be implemented as. This second technique covers both loop and queue/state-machine. Intuitively, this strategy aims to leverage the *fuzzy code comprehension* capabilities that LLMs have and infer developer intent when it comes to which code fragments implement retry. Interested readers can find more design details, insights, prompts used, and a quantitative analysis of identifying retry in large software systems in our full conference paper [211] (Sections 3.1, 3.2, 4.2, and 4.3).

4.2.1.2 Simulating a trigger exception

Once a retry location L is identified, WASABI can instrument a unit test that exercises L so that a retry-trigger exception is thrown during the test. Of course, static analysis may report that multiple exceptions could be thrown at a retry location and trigger retry, in which case WASABI creates multiple unit *retry* tests, each with one type of exception injected.

We implement this instrumentation using AspectJ, which allows us to register a handler (a *pointcut* in AspectJ), that gets executed right before a callee method (i.e., a retried method) is invoked by a caller method, both specified by us (i.e., a coordinator method). As illustrated in Listing 4.2, the handler takes as arguments the callee method name, the caller method name, and the exception to be thrown. The handler simply (1) throws the exception if this particular injection point was reached less than K times, and (2) writes a corresponding entry into the test log.

```
onCallAt(callee, caller, exception) {
  int callCount = hashTable.get({callee, caller, exception})
  if (callCount < K) {
    log.debug("Injected {exception} {callCount + 1} times, at callsite {callee} invoked from {
        caller}");
    hashTable.set({callee,caller,expcetion},callCount+1);
    throw new exception()
  }
}
```

Listing 4.2: Exception Throwing Handler Pseudocode

To decide the value of K, we consider how many retry attempts are needed to expose a retry bug. For each unit test that targets a specific retry location and retry trigger, WASABI runs it twice with two different settings of K: 1 and 100. The latter number is chosen to safely exceed all application-configured thresholds. This exercises different aspects of retry: with $K = 1$, the unit test can exercise the code after the retry and get a chance to expose HOW retry bugs—e.g., incorrect state clean-up [211]; with $K = 100$, the unit test can exercise the retry cap and delay mechanism, yet the application code after the retry logic may not execute if the application times out.

4.2.1.3 Retry test oracles

Every unit test comes with test oracles often in the form of `assert` statements, so that software bugs can manifest as test failures. However, retry bugs may not violate existing test oracles—they might degrade application performance, but not trigger crashes; or they might lead to unexpected data corruption that does not violate existing assertions. Furthermore, a violation of existing test oracles may not reflect true bugs under our exception injection scheme, as we will explain later.

To tackle this challenge, we designed three retry-specific, application-agnostic test oracles.

"Missing cap" oracle. This oracle classifies test runs based on the number of retry attempts, and is meant to catch "cap problems" of WHEN-retry bugs. In theory, missing cap retry bugs are characterized by an unbounded number of retry attempts [211]. In practice, retry caps are typically no more than 20 retry attempts or 10 minutes [55, 56, 57, 58, 59, 60, 61, 62]. In the current prototype of WASABI, each test-run log is examined to see if any fault injection handler (Listing 4.2) has either thrown 100 exceptions or ran for more than 15 minutes. If so, a missing-cap bug is reported.

Rare cases where a callee method is invoked 100 times for reasons other than retry (e.g., handling different tasks) could lead to false positives in WASABI. Similarly, while a unit test might correctly run for over 15 minutes, this is extremely rare as tests, in our experience, typically terminate within approximately 30 seconds.

"Missing delay" oracle. This oracle is designed to catch "delay problems" in WHEN-retry bugs [211]. It checks testing-run logs to see if there were any delays between consecutive retry attempts, and reports missing-delay bugs if there is no delay between successive retry iterations.

WASABI registers an AspectJ handler to generate a log entry together with the call stack right before every thread sleep API (i.e., `Thread.sleep`, `Object.wait`, `TimeUnit.sleep`, `TimeUnit.timedWait`, `TimeUnit.scheduledExecutionTime`, `Timer.wait`, and `Timer.schedule`). After each test, WASABI checks the log to see if there is a `sleep` call in-between two consecutive fault injections from the same retry location. WASABI compares the call stack to only consider a sleep invoked from the corresponding coordinator.

"Different exception" oracle. This oracle checks exception(s) thrown by the test code, not by WASABI handlers, and flags an execution as potentially buggy when the exception is *different* from the one WASABI injected. This oracle is designed to find `HOW`-retry bugs [211].

To maintain accuracy, this oracle intentionally avoids reporting correct behavior where, after a few retry attempts, the unit test gives up and re-throws the same exception that was initially injected by WASABI. While this causes the unit test to crash, it is typically correct behavior as it allows the upper layer to handle the error. Additionally, the oracle avoids reporting false positives resulting from our static analysis inaccuracies (§4.2.1.1): if an injected exception is not a retry trigger, the unit test will crash and throw the injected exception without being flagged as a bug.

4.2.1.4 Tests preparation

There are a few remaining roadblocks for WASABI to effectively utilize unit tests.

Restoring default retry configurations. Sometimes software engineers deliberately restrict retry in unit tests, by explicitly overriding the default configuration of the maximum number of retry attempts to 0, 1, or 2 (in about 10% of the test cases that cover retry logic based on our study). To counteract engineer-imposed restrictions on retry logic, we use a Python script to scan every unit test and pinpoint instances where retry parameters are altered. Next, we override these values with the default ones in the application's configuration files or documentation. This ensures that WASABI's retry testing accurately reflects the intended retry behavior without artificial limitations.

Fault-injection planning. A naive testing plan that injects trigger exceptions at every retry location in every unit test would lead to insufficient testing of some retry locations and redundant testing of others: (1) When a unit test contains multiple retry locations, injecting exceptions at an early location could cause later retry locations to be skipped—the coordinator method may terminate after several retry attempts at the early location. (2) One retry location and its corresponding coordinator method are often covered by multiple unit tests; repeatedly triggering retry attempts at the same location across different unit tests is often a waste of testing resources.

To improve this naive plan, before any fault injection, WASABI instruments every retry location reported by its static analysis, and runs the entire test suite once to figure out which retry locations are covered by every unit test at run time. WASABI then conducts a *test planning*, with the resulting plan being a list of {unit-test, retry location} pairs. WASABI's planning ensures every retry location that could be covered by the test suite appears exactly once in the plan list; WASABI also tries to maximize the number of unique unit tests covered by this plan, although there is no guarantee. Specifically, WASABI iterates through every unit test one by one. For each test t, WASABI identifies its first retry location ℓ, if any, that is not yet covered, adds $\{t, \ell\}$ to the plan, and moves on to the next test case. After iterating through every test case once, if there are still uncovered retry locations, WASABI iterates through every unit test again (and again), until every retry location is included in the plan.

For each pair in the test-plan list, WASABI uses AspectJ to turn a unit test into a series of unit *retry* tests that exercise retry triggered by different exceptions associated with that retry location, as discussed in §4.2.1.2.

4.2.2 Detecting retry bugs through static analysis

WASABI's unit testing depends on the quality and coverage of existing test suites. Moreover, WASABI's test oracles do not cover IF bugs. To fill in these gaps, we include static analysis routines/prompts as part of WASABI's detection suite.

WASABI tackles the WHEN and HOW retry bugs not cover by tests by combining large language model reasoning with static control-flow analysis. For WHEN bugs, it provides GPT-4 with the complete source file and a finely tuned sequence of yes/no prompts that probe missing retry cap or delay issues across both loop- and asynchronously scheduled retry constructs For IF bugs, WASABI relies on CodeQL to compute, for every exception E, the ration between the number of retry loops that throw E and the number of retry loops that actually actually trigger retry by branching back to the beginning of the loop. Retry locations whose ratios are near-uniform yet not absolute are surfaced as outliers, flagging them for software engineers to revisit their retry policy.

The interested reader can learn more about the design decisions, insights, and efficacy of WASABI's static analysis checking in the conference paper [211] on which this chapter was written (Sections 3.2, 4.2 and 4.3).

4.3 Evaluation

We evaluate WASABI on largely the same set of applications used in our issue study. The difference is that (1) we used the latest version of each application as of March 2023 when we started designing WASABI; and (2) we excluded Kafka and Spark from the set, and instead added MapReduce and Cassandra. We excluded Kafka because its retry logic is predominantly driven by error codes and application state, rather than exception handling, and hence is out of WASABI's scope. We excluded Spark, because of an incompatibility with the AspectJ Maven plugin [190]. Overall, we used 8 applications in our evaluation: Hadoop-Common (HA), HDFS (HD), MapReduce (MA), Yarn (YA), HBase (HB), Hive

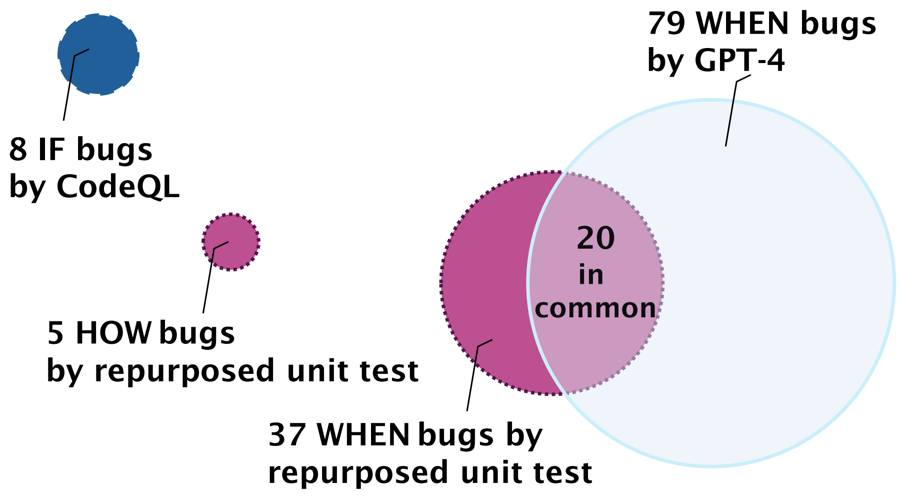

79 WHEN bugs by GPT-4
8 IF bugs by CodeQL
20 in common
5 HOW bugs by repurposed unit test
37 WHEN bugs by repurposed unit test

Figure 4.2: Bugs found by WASABI's repurposed unit testing and static checking illustrated as different circles. Our tool identifies 42 bugs through unit testing (37 "WHEN" and 5 "HOW" bugs, respectively), 79 bugs through LLM-guided static code analysis, and 8 "IF" bugs where a retry policy either should or should *not* have been implemented using control-flow analysis statistically. 20 bugs identified with the help of LLMs happen in retry structures cover by tests and can therefore be confirmed through the repurposed unit testing workflow by injecting retry-triggering exceptions. The interested reader can consult our full paper [211] for more details about WASABI's LLM-guided and statistical bug finding components.

(HI), Cassandra (CA), and ElasticSearch (EL).

We ran each application on an Ubuntu 22.04.3 LTS machine, with a 12-core Intel i7-8700 CPU, 32 GB of RAM, and 512 GB of disk space. We relied on both Java 8 and 11, Maven 3.6.3, AspectJ 1.9.19, and the AspectJ Maven plugin 1.13.1 [190]. Additionally, ElasticSearch requires Gradle 4.4.1, instead of Maven, as the build system.

4.3.1 Results on Bug Detection

In total, WASABI reports 191 *distinct* retry problems across all 8 applications evaluated, with its repurposed unit tests reporting 63 WHEN and HOW retry problems, its GPT-4 static checker reporting 139 WHEN retry problems, and its CodeQL checker reporting 9 IF retry problems. We have carefully examined every WASABI report and identified 109 of them as true bugs—WASABI unit testing has a false positive rate of 2 true bugs vs. 1 false positive, and WASABI static analysis has a rate of 1.4 true bugs vs 1 false positive. We are releasing WASABI [225] along with a detailed description of every bug reported by our tool together with our paper.

We illustrate the distribution of the 109 reports that we deem to be true bugs in Figure 4.2, and explain them in depth. Section 4.3.3 will discuss the 82 false positives in detail.

WASABI unit testing As illustrated in Figure 4.2, WASABI identifies 42 bugs through its repurposed unit testing: 20 are missing-cap WHEN problems (i.e., infinite retry); 17 missing-delay WHEN problems (i.e., aggressive retry without delay); and 5 issues related to HOW retry is implemented. Those 5 HOW-retry bugs were exposed by injecting a retry-trigger exception just once, while the other bugs were exposed by injecting retry exceptions 100 times, as discussed in §4.2.1.2. §4.3.1 shows a detailed breakdown of these bugs per target application.

As an example, after WASABI injects `SocketException` once while running *testSaveAndLoadErasureCodingPolicies* of HDFS, the test fails with `NullPointerException`. WASABI's "different exception" oracle flagged this as a potential bug. After inspection, we realized that when a transient error happens too early in function `createBlockReader`, not all objects are properly allocated. However, the `catch` block that handles `SocketException` assumes all objects were constructed and attempts to log the current program state. This logging results in a `NULL` pointer dereference.

Note that, one bug can cause multiple WASABI unit test runs to fail. For example, the above bug also caused crashes when exceptions were thrown at two other retry locations in the same retry loop. The "different exception" oracle considers two crash failures as the same bug if they have the same crash stack; the "missing cap" and "missing delay" oracles group test reports based on retry code structures (e.g., only one missing cap/delay bug is counted for each retry loop).

Without WASABI, the probability of exposing *any* of these bugs during unit testing is extremely low, if at all. Indeed, we have run the original test suite without WASABI many times, and *none* of the problematic retry logic is exercised.

Even if these bugs are exposed during testing by luck, they cannot be effectively handled by existing test oracles. The three test oracles (§4.2.1.3) are crucial for WASABI unit testing. Without these test oracles, there will be a large number of additional false negatives (i.e., all missing-delay bugs and the majority of the missing-cap bugs will be missed), and false positives. For example, about 90% of test crashes encountered by WASABI testing are caused by unit tests re-throwing the injected exceptions, and are correctly filtered out by WASABI test oracles.

WASABI static checking WASABI's GPT-4 based bug detection finds 79 WHEN bugs. Of these, 20 are also found by fault injection. Comparing GPT-4 and WASABI's repurposed unit testing, the false negatives (missed bugs) of the latter are mainly due to the lack of code coverage of existing unit tests, while the false negatives of the former are mainly due to GPT-4 struggling to reason about large files and hence not even realizing the existence of retry, which we will elaborate in the next sub-section. Also note that, although GPT-4 identified more WHEN bugs than unit testing (79 vs. 37), it also reported more false positives (60 vs. 16) due to the non-determinism of language models and their tendency to confabulate when analyzing code [120, 158, 184].

The interested reader can find the full quantitative analysis of WASABI's static checking workflow in our full conference paper [211] (Sections 4.2, 4.3).

4.3.2 Retry Code Identification and Coverage

Retry code identified As shown in Figure 4.3, WASABI identifies 323 code structures across all 8 applications where retry logic is implemented. About 70% of them are loops (i.e., 239 retry loops in total), while the rest implement retry through finite state machines

Retry Bug Type	Hadoop	HDFS	MapReduce	Yarn	HBase	Hive	Cassandra	ElasticSearch	Total
WHEN bugs: missing cap	2_1	7_2	-	1_1	13_2	3_1	1_0	1_1	28_8
WHEN bugs: missing delay	3_2	6_3	5_1	-	6_2	2_0	2_0	1_0	25_8
HOW retry bugs	-	4_2	-	-	4_2	2_1	-	-	10_5
Total	5_3	17_7	5_1	1_1	23_6	7_2	3_0	2_1	63_{21}

Table 4.1: Retry bugs reported by WASABI unit testing (subscripts: # of false positives; -: no report). WASABI indicates 63 potential bugs (28 “missing cap”, 25 “missing delay” and 10 “HOW” type). After manually analyzing each report, we determine 21 to be false positives (8 “missing cap”, 8 “missing delay”, and 5 “HOW” type). Each column shows a detailed breakdown per target application. WASABI discovers the most retry bugs for HBase (23 reported, 6 false positives), and the fewest for Elastic Search (2 reported, 1 false positives).

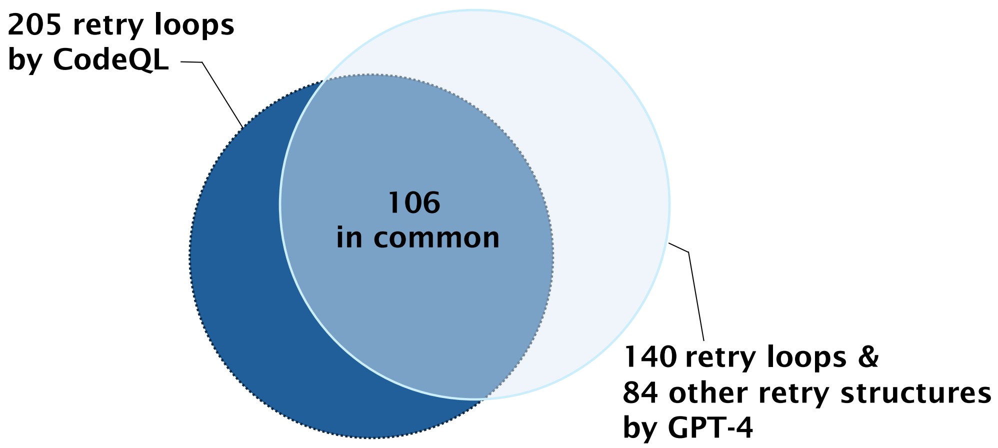


Figure 4.3: Retry code structures identified by WASABI. Our tool identifies 205 retry structures through traditional control-flow analysis using CodeQL, all implemented as loop constructs. It further identifies 224 retry structures through LLM-informed code comprehension, 140 of which are loop constructs while the remaining 84 are implemented as non-loop structures (i.e., finite state machines, asynchronous queues).

and task re-enqueuing.

Comparing the two approaches, CodeQL cannot detect non-loop retry but did manage to identify more than 85% of the retry loops reported by the two techniques. Naturally, it missed retry loops that contain no string literals, variables, or methods whose names include "retry" or "retries". GPT-4 has the advantage of identifying non-loop retry, but it missed 100 retry loops. Our investigation showed that these are located in 53 different large files. On average, these files are almost twice as large (mean: 10,539, median: 9,304 tokens) than those where GPT-4 does identify retry logic.

Both approaches occasionally mislabel locations. A manual examination of 40 sampled retry loops identified by CodeQL reveals 3 false positives: an attempt to obtain a lock and failure logging if unobtainable after n "retries"; an attempt to generate a unique string and failure after n "retries"; and token-by-token parsing of a request which may contain a "retryOnConflict" parameter. The locations found by GPT-4 have a slightly higher false-positive rate: of 100 sampled locations, we find 16 false positives, which contain re-execution behavior such as iterating through queues, or status-update polling; as well as object parsing or construction that contains a retry-named parameter. Fortunately, these false positives do not affect the accuracy of WASABI unit testing: injecting exceptions at non-retry code would cause a test to crash with the same exception, which would be pruned out WASABI's "different exception" oracle. However, the false positives in GPT-4's retry identification are connected with the false positives in its bug identification.

4.3.3 Cost and False Positives

Cost of WASABI For most of these 8 applications, WASABI unit testing took around 10 hours, with HBase taking the most time (close to 20 hours). The majority of the time is spent on running the test cases, with less than 1% spent on static analysis or post-mortem log processing. The test run time can be further broken down into two parts. First, the time to run every test in the test suite once to figure out which test case covers which retry location, as part of the WASABI test planning (§4.2.1.4). This takes 18%–32% of the total run time—all these applications come with thousands or tens of thousands of unit tests, as shown in Table 4.2, that take more than an hour to run. Second, the time to run all WASABI repurposed unit tests with injected exceptions, which takes the remainder of the test run time. Since WASABI is designed for in-house testing, we consider the overheads acceptable.

Note that, exceptions and exception handling are very costly, not to mention that one of WASABI's fault-injection policies is to throw up to 100 exceptions or terminate a unit test at 15 minutes. WASABI's repurposed unit testing only increases the original unit testing time by 2×–5×, instead of hundreds to thousands of times, because only a portion of unit tests actually cover retry locations (4%–27% across all applications as shown in Table 4.2). More importantly, WASABI's planning stage makes sure that retry locations are not repeatedly tested across different unit tests (Section 4.2.1.4), which helps cut the number of fault-injection testing runs by 27×–170×, as shown by the last two columns of Table 4.2.

App.	# Unit Tests		# Wasabi Test Runs	
	Total	CoverRetry	w/o planning	w/ planning
HA	7296	841	9156	54
HD	7642	405	7834	110
MA	1468	393	2940	48
YA	5757	764	4764	42
HB	7052	1438	4248	158
HI	35289	1505	2506	36
CA	5439	952	1132	26
EL	12045	1388	1802	28

Table 4.2: Details of Wasabi unit testing. The two "# Unit Tests" indicate the overall number of tests versus how many cover retry structures; notice that, on average, only ~10% (4%–27%) cover the application's retry logic. The next two ("# Wasabi Test Runs") indicate the number of tests Wasabi needs to run without and with test planning; notice that without the planning phase, our tool would need to run one or two orders of magnitude more tests to detect the bugs described in this section.

Cost of GPT-4. To execute the workflow that involves GPT-4, namely retry location identification and static WHEN bug detection (Figure 4.3), the median number of GPT-API calls we made for each application was about 2600 (1 call per file and follow ups). The median amount of data sent through these API calls is around 16MB and 3.3M tokens for each application. At publication time, the monetary cost of processing this volume of data using the GPT-4 API was about 8 USD per application. Costs may be further reduced through additional filtering steps, e.g., excluding from analysis files that clearly do not perform I/O.

False positives of unit testing Through the repurposed unit testing, Wasabi reported 63 bugs in total. Our investigation shows 21 of them to be false positives.

There are 5 false positives reported by Wasabi as HOW bugs. In all 5 cases, applications caught the injected exception, wrapped it inside a general exception, and the general exception led to a crash. These failures were wrongly labeled as HOW bugs by Wasabi's "different exception" oracle. Future work may prune these false positives by analyzing the exception propagation and wrapping chains.

There are 8 false positives reported by WASABI as missing-delay WHEN bugs. For all these cases, WASABI's judgment is correct—the application indeed did not take a break between consecutive retry attempts. However, our manual inspection found that the delay may not be necessary as small changes were made between consecutive retry attempts. For example, in HDFS, when a file-block fetching fails, the retry will ping a different replica node. In this case, we conservatively consider WASABI's bug report as a false positive.

Finally, there are 8 false positives reported by WASABI as missing-cap WHEN bugs. In these cases, the application actually has a cap for the number of retry attempts, and chooses to propagate the exception to an upper level after the cap is hit. However, the upper level turns out to be the test harness, which ignores the exception and continues the unit test. Making things worse for WASABI, in these unit tests, the method associated with the fault injection handler is invoked by the test harness many times to handle different tasks. As a result, the 100 retry attempts occurred across many tasks and triggered a false missing-cap bug report. Future work that makes WASABI more aware of the call context may be able to resolve most of these false positives.

4.4 Discussion

4.4.1 Mitigating false positives

Most of WASABI's unit testing false positives may be removed through further analysis of the call and exception contexts, as discussed above.

Many of the static detection false positives may be removed by collating the results of static detection with unit testing results. For code segments not covered by unit tests, other avenues for reducing false positives include: 1) appending the content of a method M callee function from a different file into the prompt referencing M, or 2) reducing the token size of large files using prompt-compression techniques [114, 115]. For the specific cases where

GPT mistakenly identifies non-retry code to be retry-related, the effect of false positives may be mitigated by presenting every bug report in two parts: which code snippet is considered as retry (easily reviewable by software engineers), and a description of the bug. We also expect the accuracy of our LLM-based static analysis to improve with future LLM models.

Note on false negatives. It is difficult to precisely measure the false negatives of a bug-detection tool. Looking at the root cause categories identify in our extend retry bug study [211], those "missing or disabled retry mechanism" bugs are not covered in the design of WASABI and will cause false negatives. Furthermore, if a bug is caused by software misconfiguration, which happens to about 10% of the cases in our dataset, it will be missed by WASABI unless the unit tests use the same mis-configured setting. A false negative could occur even for a bug whose root cause is covered by WASABI when 1) existing unit tests do not cover related code and/or 2) the retry location is missed by CodeQL and the LLM.

4.4.2 Broader system design considerations

Some design considerations would improve the quality of retry and system at large. For one, the systems we studied display an overall lack of consistency in encoding retry-errors: applications will retry based on error-code in some instances and on exceptions in others (even within the same file); wrap exceptions in an ad-hoc way; or use too-general errors, making accurate retry-or-not decisions difficult. Retry structures are also widely inconsistent—a single application might include a range of unique local implementations of queue or state-machine based retry. Reducing variance of retry structures and error definitions would help improve the correctness and maintainability of retry-related code.

Another consideration is that application testing frameworks and conventions are ill-suited to isolated and systematic retry testing. For example, tests will frequently disable retry or enforce restrictive timeouts, which are difficult to override without tedious workarounds; or use error-mitigating procedures in test harness code that conceal genuine retry problems.

More flexible testing frameworks with built-in support for retry-test configurations would reduce the burden of implementing comprehensive retry tests.

4.4.3 Relationship with prior work

Wasabi vs. fault injection tools. A wide array of tools rely on fault injection to trigger bugs in software systems, as discussed at length in §2.2. In addition, error handling, as a generic mechanism, has been investigated extensively as it is one of the main root causes of production failures. As described in §2.4.1), some works use static analysis to check faulty error propagation or code not conforming to specifications (e.g., [111]). Others, like those detailed in §2.4.2 rely on dynamic analysis and, sometimes, fault injection to trigger those type of bugs. Regardless, these techniques are *not* tailored to automatically expose retry-related bugs. Consequently, they differ from Wasabi in several aspects.

First, Wasabi faces the unique challenge of identifying retry-specific rather than general error handling problems. As discussed earlier, traditional program-dependency-based static analysis is not suitable for identifying retry functionality. Meanwhile, code constructs related to generic error handling like exception blocks yield a wide instrumentation code space where most of the code is not related to retry.

Second, by targeting retry bugs, Wasabi has different fault injection policies regarding where and what exceptions to inject as well as different test oracles than previous works. For example, prior work injects domain-specific faults like network partitions, disk faults, database read/write inconsistencies (§2.2.1); targets specific system components like cluster management controllers and microservices (§2.2.2); or focuses on certain failure scenarios like crash-recovery and crash-consistency which are typically triggered by non-recoverable instead of recoverable errors (§2.2.3). Others ask users to specify or customize injection rules (§2.2.4), instead of automating these tasks like Wasabi.

This difference also applies to recent research [40, 139], as detailed in §2.2.1 and §2.2.2.

Focusing on different types of bugs from WASABI, their design differs from WASABI's in terms of where and how to inject faults, how to judge the existence of a bug, and the types of tests used to drive fault injection.

As discussed earlier, Rainmaker [40] targets bugs triggered by transient errors in applications interacting with cloud services through REST APIs. Rainmaker perturbs requests at the HTTP boundary and reuses existing assertions in integration tests, unlike WASABI which injects exceptions at retry sites inside the application's core modules and relies on retry-specific oracles over unit tests to identify retry issues.

Legolas [229] focuses on partial failures in distributed systems, so called *gray failures*. Like WASABI, but unlike Rainmaker, it injects faults at the application level by throwing exceptions. However, it differs from WASABI in almost all other aspects of the design because the two tools target different types of bugs. Specifically, Legolas strategically injects faults to maximize a system's abstract state coverage, whereas WASABI injects faults at retry locations. Legolas uses existing system-crash and checkers specific to gray failure to judge whether a bug has occurred, unlike the retry-specific oracles used by WASABI. Similar to WASABI, Legolas needs to use system tests instead of unit tests to trigger gray failures.

Anduril [180] addresses a different stage of reliability, namely *reproducing* a known fault-induced production failure. Given a failure signature and a replayable workload, it combines static causal pruning with a feedback-driven search to throw a specific exception at a precise execution point such that the observed execution matches the target incident. In contrast, WASABI proactively triggers retry bugs before deployment that are unknown *a priori* to software engineers, by identifying retry locations and injecting exceptions while running existing tests, and uses retry-specific oracles to determine whether a bug occurred.

WASABI vs. AI-guided software testing tools. The rise of LLMs has brought opportunities to software engineering research, with an emergent set of papers applying LLMs to code generation (§2.8.2), testing (§2.8.1), repair (§2.8.3), analysis (§2.8.4), as well as summarization and documentation ((§2.8.5). WASABI is orthogonal to these works by using LLMs to pinpoint a common yet unstructured functionality—retry logic—in large codebases.

WASABI in the broader context of Cloud failures. Much work is dedicated to studying failures in cloud systems (§2.1). Some papers introduce new taxonomies for bugs in cloud and distributed systems [80, 90, 155], while others focus on new or understudied classes of bugs like metastable failures [28, 105], cancellation issues [202], or cross-system defects [217]. Our paper sheds light on retry bugs and provides insights on why retry functionality is difficult to implement, test, analyze, and reason about.

4.5 Summary

Retry had long served as the last line of defense against transient faults, yet a recent study [211] showed that faulty retry logic frequently amplified failures by crashing even when recoverable errors occur, exhausting resources, or corrupting program state. The study found that correctness hinged on three intertwined decisions: IF a task should retry, WHEN and how often it should retry, and HOW to execute the retry without harming system state. Unfortunately, software engineers routinely mishandled all three. The absence of a common standardized retry libraries and the prevalence of diverse retry implementations, from simple loops, to asynchronous re-enqueues, to state-machine workflows, rendered conventional program analyses ineffective to detect those bugs, while resilience libraries and testing frameworks leave most retry paths unexamined.

To solve some of these challenges, this chapter introduced WASABI, a toolkit that combines static program analysis, LLM-guided code reasoning, fault injection, and unit testing to tackle retry-related issues in software systems. Our toolkit addressed policy mistakes, in particular *when* to retry, and mechanism errors like *how* the retry executed, across loops, asynchronous queues, and state-machine retry constructs.

We showed how WASABI successfully repurposes existing unit tests into bug-triggers, by injecting candidate retriable exceptions at retry sites identified through static and LLM-guided analysis, and forcing rarely executed retry paths without requiring new test cases. Specialized oracles then label faulty executions as retry bugs based on the patterns reported by users [211]. Overall, WASABI identified over 100 distinct, previously unknown retry bugs in 8 Java applications across all three types of common root causes [211]. In particular, WASABI identified 42 retry bugs by reutilizing existing unit tests.

This chapter highlights the potential of combining complementary static, dynamic, and AI-guided approaches to identify and improve bug-finding at the mechanism level. As discussed earlier, these issues surface because of faulty implementation sometimes often on top of faulty timing and policy-level behavior which needs to be considered holistically for effective bug diagnosis.

CHAPTER 5
ANALYSIS AT THE MODULE LEVEL: PREDICATE-DIRECTED TEST SYNTHESIS FOR IMPROVING CODE EFFICIENCY

This chapter explores how to efficiently apply the two principles introduced previously by showing how AI-guided fuzzing can help isolate performance bottlenecks that require bug-finding at the *module level* granularity.

Analyzing and optimizing the performance of software systems requires both (i) locating cost-apmlifying code paths, namely those whose resource cost increases drastically on specific inputs, and (ii) generating inputs or workloads that exercise those paths. On the one hand profiling and analysis tools record the workload provided, typically small and correctness-oriented tests found in existing test suites. Their measurements, therefore, seldom indicate bottlenecks that surface only on worst-case inputs. On the other hand, fuzzers can generate those worst-case inputs, yet only when software engineers annotates specific program statements that guides the fuzzer toward revealing a performance issue (e.g., adding an associated "cost" like execution frequency). With the increasing adoption of AI in software development, recent work explores using LLMs to overcome these challenges and directly synthesize performance stressing inputs. However, current LLM-guided test generators merely increase input length, missing semantic triggers that are more likely to reveal more nuanced inefficiencies and optimization opportunities.

To mitigate these shortcomings we present Wedge, a novel LLM-guided contrasting execution analysis that can better guide test generation tools. Wedge synthesizes explicit performance-characterizing constraints in the form of branch conditions to partition the execution space of a target program into performance-specific regions. Our framework relies on the fuzzy code-comprehension capabilities of LLMs to generate performance constraints as an intermediate reasoning step. These constrains are then integrated into the target programs as branch instructions that a fuzzing tool can explore. Thus, when integrated with a coverage-guided fuzzer, reaching different regions introduces explicit rewards for test generation to explore inefficient implementations.

Our evaluation shows that WEDGE introduces a significant slowdown compared to tests in CodeContests [149], a state-of-the-art C/C++ benchmark used to evaluate LLM-guided test generators. It also provides a better running time contrast between original programs and their corresponding versions optimized by state-of-the-art LLM-guided techniques. From an utility perspective, integrating our synthesized tests substantially improve the existing code optimization approaches that rely on test-driven execution feedback. We release PERF-FORGE [1], the performance tests generated by WEDGE, to support detecting performance bottlenecks and to help benchmark future approaches for efficient code generation.

This chapter is, in part, a reprint of the material currently under submission at a premiere machine learning conference [236]. This chapter focuses on how can how AI approaches help identifying bottlenecks that lead to performance issues and, more broadly, how to improve performance testing in software systems through LLM-informed It provides a high-level level overview of the synergy between AI-guided strategies and fuzzers. Consequently, it only offers a glimpse of the low-level design decisions that WEDGE makes in order to re-engineer an off-the-shelf fuzzer, AFL++, to synthesize performance-exercising tests. The interest reader can find those design details, insights, and additional quantitative analysis in the full manuscript [236].

5.1 Motivation

Analyzing and optimizing the performance of software systems relies on two complementary tasks. First, performance analysis techniques must identify the code paths whose resource

1. https://github.com/UChiSeclab/perfforge

cost grows disproportionately under certain inputs. These paths arise from algorithmic worst-case behavior (i.e., increased CPU utilization), memory inefficient utilization, excessive I/O, or cascading bottlenecks [146]. Second, techniques must synthesize concrete inputs or workloads that drive execution through those paths so that slowdowns become observable during automated evaluation.

Traditional program analysis and profiling techniques struggle when those workloads are missing [146]. Many approaches rely on manually crafted stress tests or, worse, on inputs originally written only to check correctness. Such tests rarely trigger the costly branches that reveal inefficiencies, leaving software engineers without any hint of potential issues.

This limitation becomes even more pronounced when optimization relies on large language models (LLMs). LLMs have shown increasing promise in optimizing code efficiency beyond compiler techniques [23, 47, 65, 101, 103, 148, 157, 185, 205]. Moreover, as some approaches integrate execution feedback to further optimize the code [101, 185, 239], running performance stressing tests could incrementaly reveal more precise optimization opportunities by exposing performance bottlenecks iteratively. So, to understand whether those synthesized optimizations are meaningful or use them as feedback to further optimize the code, there is a growing need for high-quality stress inputs that are capable reveal subtle performance bottlenecks efficiently and effectively. To illustrate this, consider a program that computes Fibonacci sequences. An optimization from recursion to iteration in Fibonacci number calculation incurs only a negligible performance improvement when evaluated with a default test ($n = 3$) that focuses on testing correctness, while a performance-stressing input ($n = 40$) reveals the orders (10^6) of the larger gap. Consequently, providing the appropriate performance stress input is crucial to accurately reflect this performance gap.

While generating correctness tests has been extensively studied in the past [12, 74, 86, 94, 95, 107, 168, 178, 197, 223, 238], there is far less research available for performance stress test generation [137]. This is also the status quo for LLM-based gest generation techniques.

Existing tests in common benchmarks used to evaluate LLM-guided code optimizations like those described in §2.8.6 (e.g., HumanEval [38]), were shown to have limited scope and low complexity and thus fail to adequately stress the code performance against more demanding conditions [159]. As a result, they are also more susceptible to the noise introduced in the execution environment, thus failing to reliably quantify the optimization and reveal insightful optimization opportunities.

Typically, fuzzing is a popular technique that has been successfully applied to generating performance-stress. Tools such as PerfFuzz [137], FuzzFactory [179], or SlowFuzz [188] demonstrate that guided mutation can discover non-obvious worst-case inputs, yet they require a signal, like a branch, predicate, or cost proxy, to tell the search that it is on the right track. Generating high-quality signals automatically remains an open challenge.

With the increasing capabilities of LLMs, other recent works are leveraging LLMs by prompting them to generate test generators [101, 159]. For example, EvalPerf [159] introduced a scale parameter to control the input size, with the assumption that it is the key determining factor to performance-stressing. However, such a biased preference over large tests misses the opportunity to reason about their relationship to inefficient program behaviors beyond the size. For example, calling quicksort can suffer from the suboptimal performance [187] when its input is reversely sorted ($O(n^2)$ in the worst case). When two inputs are both at the maximum length `n`, the reversely sorted one is more stressing than another randomly ordered one ($\mathcal{O}(n \log n)$) on average.

Our approach. In this context, the current chapter tackles the first complementary task mentioned in the beginning, namely identifying the cost-amplifying code paths themselves. Specifically, we design WEDGE, a framework that generates performance test inputs beyond simply stressing the sizes. Our key insight is that the limitation of LLMs in generating performance-stressing tests boils down to the inherent challenge of connecting the *local* performance-related program behavior all the way back to the program inputs [116], while directly reasoning about the local behaviors is comparatively easier.

For example, we can easily specify the local variable `arr`, the argument to a `quicksort` deeply nested inside the program, to be *reversely sorted* to trigger its inefficient behavior, while predicting what program inputs lead `arr` to be reversely sorted is more challenging as that requires reasoning about the control and data flow based on the precise understanding of the program semantics. Such reasoning is extremely challenging due to an impractically large search space, e.g., tracking a combinatorial number of program paths [32, 45, 116, 124].

Based on such insight, WEDGE alleviates the reasoning task of LLM on performance-related behavior by asking it to synthesize *the performance-characterizing constraints* as condition checkers, e.g., `all(l[i] > l[i+1] for i in range(len(l)-1))`, and instrument the program with these checkers at the appropriate program points. WEDGE then leverages the coverage-guided fuzzers, the search-based testing technique [74, 78] with the goal to maximize the code coverage, to scale test input generation that sidesteps the expensive iterative queries to LLMs. As the inputs achieving new coverage are rewarded and prioritized in the fuzzer, checker branches inserted by WEDGE serve as the coverage signal to bias the fuzzing to generate likely stressing inputs more efficiently. Because the resulting checkers are ordinary branches, they can be consumed unchanged by high-scale fuzzing tools such as Nyx [197], Jackalope [84], or Syzkaller [85], giving Wedge immediate reach to code bases far larger than our prototype's context window.

To enhance performance constraint reasoning, we develop a reasoning template that elaborates on the procedures to contrast the pair of disparate execution profiles to gain insight into inefficient implementations. We then instruct LLM to reason about performance constraints (in natural language and code) in multiple phases to localize the appropriate program points and implement the corresponding constraint checkers. Besides guiding the fuzzer using constraint checkers, WEDGE further accelerates the input search by replacing the fuzzer's default input mutator [74] with a constraint-aware one that steers the input

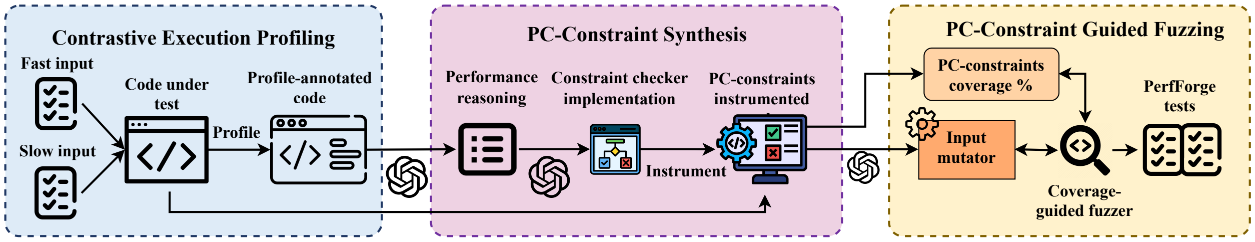


Figure 5.1: Workflow of WEDGE. First, our tool profiles the code-under-test to identify a pair of inputs with contrastive execution profile ("fast" vs "slow" execution). Second, with this information, it asks a LLM to infer performance-characterizing constraints and instrument the code with checkers. Third, it runs the instrumented code through a customized fuzzing tool to find performance-stressing inputs.

mutation towards likely constraint-satisfying inputs, while also enforcing the mutation to respect the input grammars [153, 203, 209, 241]. Figure 5.1 presents our workflow (see detailed description in Section 5.3).

Results. Our extensive evaluation shows that the tests generated by WEDGE are substantially more performance-stressing than the default ones in the existing benchmark and those generated by the state-of-the-art [159] by 84.5%. With more stressing tests, WEDGE precisely pinpoints the potential inefficient implementations and thus introduces approximately 10 percentage points more efficiency improvement on the generated code than that of default tests when used to guide the iterative code optimization approaches via test-driven execution feedback [101]. Our ablations confirm the effectiveness of the synthesized constraints in guiding the fuzzing and input mutation, i.e., achieving 4× improvement over plain fuzzing using AFL++. In addition, we show that the generated constraints effectively characterize the performance, where the constraint-satisfying inputs are 38.6× slower than constraint-violating inputs.

5.2 Overview

We start with discussing the related works on code efficiency evaluation and stress test generation. We then use a motivating example to demonstrate the advantage of Wedge over the existing approaches.

5.2.1 Relationship with prior work

While traditional AI-guided code generation primarily focused on generating correct code [13, 15, 38, 97, 158, 249], there are growing efforts to generating efficient code beyond correctness [102, 159, 186, 205]. However, existing efficient code generation techniques still largely rely on correctness tests to evaluate performance improvements [185, 205] which typically cannot faithfully measure them [102, 103, 159, 186]. Some of those tools rely on test execution feedback driven to further optimize the code [103, 185]. These approaches can miss optimization opportunities when tests do not reveal the performance bottleneck (as shown in Section 5.4.3).

To address these challenges, recent works have focused on performance test generation to benchmark efficient code generation as discussed in §2.8.1. However, these approaches either generate infeasible or inefficient inputs and thus rely on manual correction, or their task formulation often prevents the LLMs from reliably reasoning about the program behavior, i.e., by directly prompting them to generate the stressing inputs based on the full code snippets. With such a nontrivial task, LLMs have to identify the inefficient implementation, understand the run-time behavior to exercise it, and reason all the way to program input. Therefore, they often end up taking reasoning "shortcuts" and reduce to only generating length-stressing inputs that fail to reveal more intricate/complex inefficient implementation.

Performance testing has been studied extensively before LLM-based approaches, in particular relying on fuzzing (§2.3.2. Traditional techniques rely primarily on static or dynamic analysis (e.g., fuzzing). However, some suffer from scalability issues such as path explosion, fine-grained performance profiling overhead, and the lack of oracles to precisely capture inefficient behaviors or symptoms.

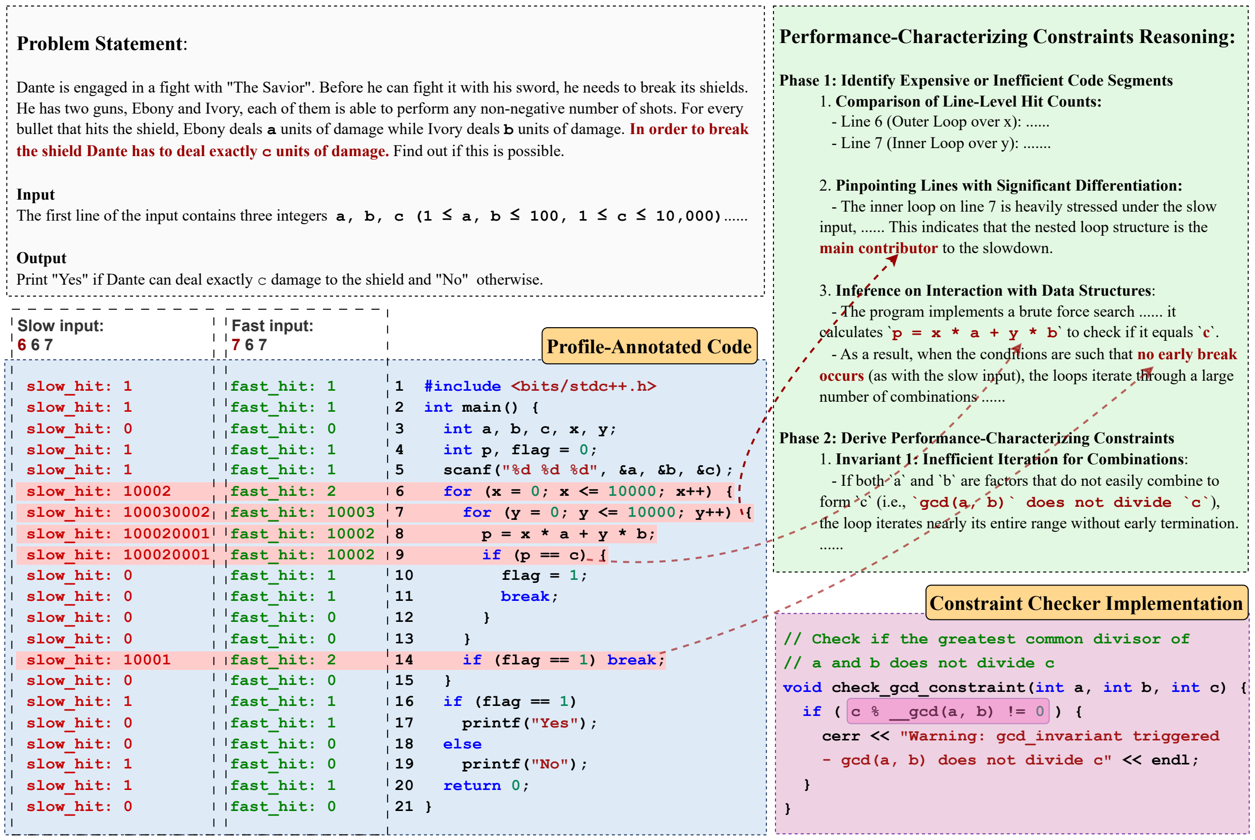
Our work aims to support fuzzers overcome those challenges by providing better signal to indicate their mutation or search components that the generation is making progress towards exercising cost-amplifying code paths. Additionally, WEDGE restricts LLMs to focus specifically on local performance-characterizing constraints to avoid overly relying on reasoning globally about to the input, and leverages the guided-search of fuzzing to reach those constraints gradually. Therefore, it captures the inefficient behaviors without expensive profiling and enables the test generation beyond length-stressing.

Finally, a significant number of papers focus on performance bug-finding (§2.6) or on building efficient software tracing infrastructure for locating such defects (§2.7.2). WEDGE complements those efforts since it aims to support bug detection and diagnosis by providing efficient inputs that exercise bottleneck code.

5.2.2 *Motivating Example*

Let us consider an example from CodeContests, Codeforces problem 633A [51] (gray box, Figure 5.2). Given three integers a, b, and c ($1 \leq a, b \leq 100, 1 \leq c \leq 10,000$), the goal is to decide whether there exist non-negative integers x, y such that $a \cdot x + b \cdot y = c$. This problem is classically known as finding solutions to a two-variable linear Diophantine equation [228].

The code snippet (blue region) shows an implementation that solves this problem. The code systematically tries every pair of values by iterating two nested loops over fixed upper bounds of $10,000$ (lines 6–7), computing the value of the Diophantine equation, and checking whether it is equal or exceeds c. Because the loops use fixed upper bounds rather than adapting to the value of c, the code could examine nearly the entire $10,000^2$ value space. Apart from skipping sums greater than c or breaking once a match is found (lines 9-11), the code bears the full brute-force cost.



Problem Statement:

Dante is engaged in a fight with "The Savior". Before he can fight it with his sword, he needs to break its shields. He has two guns, Ebony and Ivory, each of them is able to perform any non-negative number of shots. For every bullet that hits the shield, Ebony deals `a` units of damage while Ivory deals `b` units of damage. **In order to break the shield Dante has to deal exactly `c` units of damage.** Find out if this is possible.

Input
The first line of the input contains three integers `a, b, c (1 ≤ a, b ≤ 100, 1 ≤ c ≤ 10,000)`......

Output
Print "Yes" if Dante can deal exactly `c` damage to the shield and "No" otherwise.

Slow input:
6 6 7

Fast input:
7 6 7

Profile-Annotated Code

```
slow_hit: 1          fast_hit: 1       1  #include <bits/stdc++.h>
slow_hit: 1          fast_hit: 1       2  int main() {
slow_hit: 0          fast_hit: 0       3    int a, b, c, x, y;
slow_hit: 1          fast_hit: 1       4    int p, flag = 0;
slow_hit: 1          fast_hit: 1       5    scanf("%d %d %d", &a, &b, &c);
slow_hit: 10002      fast_hit: 2       6    for (x = 0; x <= 10000; x++) {
slow_hit: 100030002  fast_hit: 10003   7      for (y = 0; y <= 10000; y++) {
slow_hit: 100020001  fast_hit: 10002   8        p = x * a + y * b;
slow_hit: 100020001  fast_hit: 10002   9        if (p == c) {
slow_hit: 0          fast_hit: 1       10         flag = 1;
slow_hit: 0          fast_hit: 1       11         break;
slow_hit: 0          fast_hit: 0       12       }
slow_hit: 0          fast_hit: 0       13     }
slow_hit: 10001      fast_hit: 2       14     if (flag == 1) break;
slow_hit: 0          fast_hit: 0       15   }
slow_hit: 1          fast_hit: 1       16   if (flag == 1)
slow_hit: 0          fast_hit: 1       17     printf("Yes");
slow_hit: 0          fast_hit: 0       18   else
slow_hit: 1          fast_hit: 0       19     printf("No");
slow_hit: 1          fast_hit: 1       20   return 0;
slow_hit: 0          fast_hit: 0       21 }
```

Performance-Characterizing Constraints Reasoning:

Phase 1: Identify Expensive or Inefficient Code Segments

1. **Comparison of Line-Level Hit Counts:**
   - Line 6 (Outer Loop over x): ......
   - Line 7 (Inner Loop over y): ....

2. **Pinpointing Lines with Significant Differentiation:**
   - The inner loop on line 7 is heavily stressed under the slow input, ... This indicates that the nested loop structure is the **main contributor** to the slowdown.

3. **Inference on Interaction with Data Structures**:
   - The program implements a brute force search ...... it calculates `` `p = x * a + y * b` `` to check if it equals `` `c` ``.
   - As a result, when the conditions are such that **no early break occurs** (as with the slow input), the loops iterate through a large number of combinations ......

Phase 2: Derive Performance-Characterizing Constraints

1. **Invariant 1: Inefficient Iteration for Combinations**:
   - If both `a` and `b` are factors that do not easily combine to form `c` (i.e., `` `gcd(a, b)` does not divide `c` ``), the loop iterates nearly its entire range without early termination.
   ......

Constraint Checker Implementation

```
// Check if the greatest common divisor of
// a and b does not divide c
void check_gcd_constraint(int a, int b, int c) {
  if ( c % __gcd(a, b) != 0 ) {
    cerr << "Warning: gcd_invariant triggered
    - gcd(a, b) does not divide c" << endl;
  }
}
```

Figure 5.2: Motivating example from Codeforces (prob. 633A, sol. 622) showing how WEDGE reasons about and generate performance-characterizing constraints, and implements corresponding checkers.

The right half of Figure 5.2 (green and purple boxes) shows how WEDGE infers performance-characterizing constraints specific to this program, inspired by the contrasting execution traces that share similar inputs but have disparate behavior (manifested by the per-statement execution counts). In particular, our tool identifies specific relations among the local variables `a,b,c` to stress the nested loops to exhaust their maximum iterations. The green box shows the LLM's reasoning process, while the purple box shows the performance-characterizing constraints synthesized as a C++ checker by the LLM to be instrumented in the program.

Our key observation is that these constraints are more local, fine-grained, easier to generate, and cannot be captured by state-of-the-art techniques (e.g., [159, 186]), which focus primarily on maximizing the input values and size. Therefore, such performance-stressing constraints serve as more appropriate interfaces for LLMs to communicate their reasoning to the existing test generation tools than directly asking them to generate performance-stressing inputs.

5.3 WEDGE Framework

This section elaborates on the key components in WEDGE as presented in Figure 5.1.

Problem Statement We formally define the performance-stressing test generation problem. Given a program $\mathcal{P}$ accepting a valid input (conforming to a validity specification $\mathcal{V}$) , the set of all valid inputs is denoted as $\mathcal{I}_{\mathcal{V}}$. With a valid input $\forall i \in \mathcal{I}_{\mathcal{V}}$, the execution of $\mathcal{P}$ (denoted as $E_i = \mathcal{P} \cdot i$) yields an execution time [2] T_i. The goal of stress test generation is to generate a subset of valid inputs $I^* \subset \mathcal{I}_{\mathcal{V}}$, such that the average execution time of I^* is maximum.

At a high level, WEDGE takes as inputs a coding problem statement $\mathcal{S}$, a correct solution program $\mathcal{P}$, a set of default correctness tests $\mathcal{I}_D$, and an Large Language Model (LLM), and produces a set of performance-stressing test inputs I: $I = \text{WEDGE}(\mathcal{S}, \mathcal{P}, \mathcal{I}_D, \texttt{LLM})$.

5.3.1 Contrastive Execution Profiling

We first collect high-quality contrastive execution feedback from fast and slow executions to facilitate reasoning about performance-characterizing constraints. This is achieved in two steps.

Contrastive input pair mining. In this step, WEDGE runs $\mathcal{P}$ against a set of user-provided tests $\mathcal{I}_{\mathcal{D}}$, e.g., existing correctness tests provided by the dataset to mine a con-

2. For clarity, we use "execution time" here to represent the execution cost, but in experiments we use the number of CPU instructions. See our justification in §5.4.1

trastive (slow, fast) input pair (i_{slow}, i_{fast}). During test execution, WEDGE collects the execution cost of each input, measured by the number of executed instructions (denoted $|I|$ in our experiments). WEDGE then mines the contrastive input pairs based on the two metrics: (1) similarity defined as the sum of the match ratio (i.e., the number of common array elements divided by the length of the shorter array) and the Jaccard similarity [176], and (2) execution cost ratio, defined as the ratio of the slow input's cost $|I|_{slow}$ to that of the fast input $|I|_{fast}$. Input pairs are then ranked based on their similarity and execution cost ratio, and WEDGE selects the top-ranked pair as the contrastive input pair (i_{slow}, i_{fast}).

Profiling feedback collection. WEDGE executes $\mathcal{P}$ with i_{slow} and i_{fast}, collecting execution feedback (coverage and hit count) F_{slow} and F_{fast}. Considering such a contrastive execution pair provides the key behavior insight [150], we prompt LLM to pinpoint the differences to reason why one input leads to significantly slower execution.

5.3.2 Performance-Characterizing Constraint Synthesis

WEDGE generates the constraints in two steps: it initially generates the constraints $\mathbb{C}$ in natural language, then prompts the LLM to implement the corresponding constraint checkers and insert them to the fuzz driver $\mathcal{P}$ to produce the instrumented fuzz driver $\mathcal{P}'$.

Performance-characterizing constraint reasoning. A constraint is a predicate on the program state (e.g., variable values) and expressed as a conditional statement, e.g., `if (n > 1)`. Given a performance-characterizing constraint c and a given set of inputs $\mathcal{I}_{\mathcal{V}}$, some inputs may satisfy the constraint while others may not. We denote them as $\mathcal{I}_{\mathcal{S}}$ and $\mathcal{I}_{\mathcal{N}}$, respectively, where $\mathcal{I}_{\mathcal{V}} = \mathcal{I}_{\mathcal{S}} \cup \mathcal{I}_{\mathcal{N}}$, and the corresponding average execution time $\overline{\mathcal{T}_{\mathcal{S}}} > \overline{\mathcal{T}_{\mathcal{N}}}$.

WEDGE first constructs a comprehensive *performance reasoning prompt template* that contains the problem statement $\mathcal{S}$, solution program $\mathcal{P}$, contrasting input pair (i_{slow}, i_{fast}), the profiling feedback information F_{slow} and F_{fast}, and multiple manually-crafted constraints as few-shot examples. The performance constraint reasoning technique can be denoted as: $\textit{ReasonPerf}(\texttt{LLM}, \mathcal{S}, \mathcal{P}, (i_{slow}, i_{fast}), (F_{slow}, F_{fast})) = \mathbb{C}$, where $\mathbb{C} = \{c_i\}_{i=1}^{N}$ is a set of generated constraints. The template explicitly instructs the LLM to reason about performance constraints in multiple phases, as shown in Figure 5.2. In Phase 1, the LLM needs to identify expensive or inefficient code segments. This includes: 1) comparing line-level profiling information, e.g., hit counts, between the fast and slow runs, 2) pinpointing lines or functions that get significantly more hits under the slow input, and 3) inferring how these lines might interact with data structures, loops, recursion, etc., especially as they relate to the input constraints (e.g., n <= 100). In Phase 2, the LLM will derive performance-characterizing constraints in natural language. By enforcing the LLM to reason about the constraints with Chain-of-Thought prompting [226], WEDGE collects insights into performance and generates high-quality constraints $\mathbb{C}$ (Figure 5.2 green part).

Box 1: Main prompt used by WEDGE

(A) Context

You are an experienced C software engineer focusing on performance bottlenecks. You have:

1. A problem statement describing a task or algorithm (with constraints such as $n \leq 100$).

2. A C program that implements a solution to that problem.

3. Two inputs: a "fast" input that completes quickly, and a "slow" input that takes much longer—both inputs have similar size/structure.

4. Line-level hit counts for both runs, showing which lines get hit significantly more often on the slow input.

Your goal is to diagnose why the program runs slowly for the slow input and derive conditions or invariants that capture what triggers this slowdown.

(B) Tasks: Analyze the given code and generate performance-characterizing invariants in natural language

Phase 1: Identify expensive or inefficient code segments.

continued on next page

continued from next page

1. Compare line-level hit counts between the fast and slow runs.

2. Pinpoint lines or functions that get significantly more hits under the slow input.

3. Infer how these lines might be interacting with data structures, loops, recursion, etc., especially as they relate to the input constraints (e.g., $n \leq 100$).

Phase 2: Derive performance-characterizing invariants (natural language).

1. Generate natural language statements that describe conditions under which the program likely enters a slow path.

2. Avoid using specific numeric values from the slow input; abstract them into categories or thresholds. However, make sure those thresholds adhere to the input constraints of the problem.

3. Correlate these conditions strongly to input patterns (e.g., "when n is close to 100 and there is a nested loop," or "when a data structure is repeatedly sorted").

4. Ensure your statements are broad enough to catch possible future slow scenarios, but still reflect realistic triggers given the constraints (like $n \leq 100$).

Note that not all performance-characterizing invariants are about maximising input size. You may refer to the following examples for inspiration — some maximising the input size, some not — but do not simply replicate them exactly. Rather, use them as inspiration to infer and tailor performance-characterizing invariants tailored for the C code and problem statement you were asked to analize:

(Include the same examples you have, with indentation or
to split lines appropriately.)

(C) Output Requirements

1. Provide a list of natural language performance invariants explaining under what conditions the code slows down.

2. Do not mention or rely on exact values from the provided slow input.

continued on next page

continued from next page

3. Use or suggest threshold values that align with problem constraints (e.g., $n \leq 100$).

4. The output should be a concise, descriptive set of statements about performance triggers.

(D) Important Considerations

1. Avoid hardcoding. Don't rely solely on the exact values from the provided slow input; think in terms of categories or thresholds that lead to slow execution.

2. Avoid checks inside tight loops. Place checks in a way that does not significantly degrade performance.

3. Focus on fuzzer utility. The checks should help a fuzzer detect slow performance triggers by hitting these conditions.

(E) Problem Statement

`problem_statement`

(F) Program Solving the Problem Statement

`one_solution`

(G) The Slow & Fast Inputs

(G.1) Slow Input

`slow_input`

(G.2) Fast Input

`fast_input`

(H) Hit Count Information of Slow Input and Fast Input (Aggregated):

`product_cov`

Constraint checker implementation. WEDGE prompts the LLM with the constraints $\mathbb{C}$ and instructs it to implement the checker code faithfully and produce the instrumented program. The instrumented program with inserted checker code $\mathcal{P}'$, will be used as the target program to fuzz: $\mathcal{P}' = \mathit{Instrument}(\texttt{LLM}, \mathcal{P}, \mathbb{C})$.

5.3.3 Performance-Characterizing Constraint Guided Fuzzing

In this stage, WEDGE launches coverage-guided fuzzing against the instrumented program $\mathcal{P}'$ to search for constraint-satisfying inputs.

At a high level, WEDGE uses AFL++ as its fuzzing engine, however relying on default mutators are ill-suited for generating performance-exercising inputs. The key shortcoming is that default mutators are optimized for *binary fuzzing* through random bit flips. Thus, these mutators largely generate invalid inputs that breach specification (problem statement) constraints. To tackle this, WEDGE prompts the LLM with the constraints, specifications, and code examples to synthesize an input-grammar performance-characterizing constraints aware mutator. Equipped with custom mutators, WEDGE lanches a fuzzing campaign on the original program instrumented with performance-characterizing constraints, collects all generated inputs, and selects those top ten that induce the longest execution in terms of instruction count, per program. Interested readers can find a more detailed description of WEDGE in the pre-print manuscript on which this chapter is based on [236].

The set is then collected into a high-quality of performance-exercising test harness, called PERFFORGE [3].

5.4 Experiments

5.4.1 Setup

Test generation baselines. We evaluate PERFFORGE tests against four benchmarks. The first two serve to compare the efficacy of our performance-stressing inputs. Specifically, we consider (1) EVALPERF [159], which uses LLMs to synthesize a parameterized

3. https://github.com/UChiSeclab/perfforge

input generator and progressively scales input size until a predefined timeout or out of memory. Since our dataset lacks canonical reference implementations, we consider two variants: EvalPerf$_{\text{SLOW}}$, which uses the slowest solution as the reference, and EvalPerf$_{\text{RAND}}$, which uses a random solution. The second baseline is *TG-prompt* following the recent work [65, 102, 186] that instructs an LLM to directly synthesize the performance test generator given the problem specification and its constraints.

Utility baseline. To measure the utility of our generated tests, we evaluate existing code optimization approaches based on LLM code generation. We consider running PerfForge against Pie [205], an LLM-based code optimization that fine-tunes the LLM on slow and fast code pairs, which relied on correctness tests to evaluate its performance improvements.

Metrics. We primarily rely on CPU instruction count to measure the effectiveness of PerfForge tests, considering it is more stable across runs, platform-agnostic, and strongly correlates with performance bottlenecks [48, 52, 108, 247]. In contrast, physical running time offers an intuitive measure, it is susceptible to interference from transient system effects—such as background processes, scheduling variability, and I/O fluctuations—which can obscure true computational cost [49, 247]. To further reduce the noise, we average the CPU instructions over *five* runs for each program throughout all experiments.

Recently, LLM-based code analysis tools started to rely on instruction count measured through hardware counters [159, 186] or emulation [205] to evaluate code efficiency as it provides a more reliable, low-variance measurement than physical time alone. Recently, experiments in [186] found instruction count measurements are approximately 1000$\times$ magnitude more stable than physical execution time. This mirrors our own findings: our experiments reveal that physical time is $\sim$400$\times$ more variable than CPU instruction counts (12% versus 0.03%, on average, see §5.4.2).

Some prior work (e.g., [205]) rely on CPU simulators like Gem5 [24]. While Gem5 is known to be stable, its overhead is significant, and the CPU simulator does not necessarily reflect physical performance [24].

Dataset. We evaluate WEDGE on CodeContests [149] with a wide range of competitive programming problems and human-written solutions. Test cases include the default inputs from the original open-judge platforms as well as additional inputs generated by the authors [149]. We largely focus on C++ solutions to ensure comparable measurements, with a small subset of Python programs for the usefulness investigation (§5.4.3). We rank the problems based on the coefficient of variation [159] of the CPU instruction counts and select the top 300 problems. This ensures the selected problems feature diverse solutions and potentially have enough room for optimizations for part of the solutions. WEDGE generates tests for 207 of them, but after excluding those where baselines cannot produce valid inputs, we arrive at 154 problems and 33,020 C++ programs.

Note that the original CodeContest dataset contains over 3,000 problems with hundreds C++ solutions each. It is thus infeasible to evaluate that many executions due to both computational and monetary costs. To address this, we focus on a smaller yet meaningful subset by applying the following filtering criteria.

(1) *Sufficient computation.* We remove problems whose solutions never exceed 100,000 instructions on any input to reduce the impact of potential noise following [159].

(2) *Sufficient solutions.* We remove problems with fewer than 10 correct solutions because we need to run fuzzing on a reasonable amount of correct solutions to obtain meaningful measurements.

(3) *Sufficient test inputs.* We remove problems with fewer than five default tests because we need enough inputs for fuzzing to be effective and also to identify contrasting input pairs (§5.3.1).

(4) *Single output.* We filter out problems that accept multiple correct outputs.

(5) *Diversified performance.* We only include problems with diversified performance across the solutions. Specifically, we use the Coefficient of Variance (CV) to measure the diversity,

following [159]. Low diversity means most solutions have similar performance, indicating the solutions are likely optimal or close to optimal. Intuitively, there are fewer opportunities to identify performance-characterizing constraints on the optimal solutions.

Fuzzing and input filtering. To collect inputs, We run WEDGE's fuzzing (based on our modified AFL++) for one hour for each solution in parallel. Not all generated inputs strictly conform to the validity constraints $\mathcal{V}$ (§5.3) [159]. WEDGE applies a two-stage automatic filter to filter out likely invalid inputs. WEDGE first prompts an LLM to generate the validator based on the problem statement and use the official tests in CodeContests to check the validity of the validator. WEDGE then checks the output consistency across different solutions (labeled correct in CodeContests) under the same input, following the existing work [149]. Any input leading to inconsistent outputs will be filtered out (detailed in [236]). After these, we rank the tests for each solution in the dataset based on the slowdown they introduce. We then select the top ten longest-running tests for each program and aggregate them as part of our benchmark, PERFFORGE.

5.4.2 Main Results

To evaluate the efficiency of WEDGE and its accompanying PERFFORGE benchmark we run each solution for each problem against the generated tests and compare the number of instructions executed against the EVALPERF and TG-prompt baselines. To reduce noise, we run each program five times, and compute the average instruction counts for those runs. We then compute the average of those values across all programs and all problems.

As Table 5.1 shows that tests generated by WEDGE lead programs to execute, on average, 84.5% and 85.7% (70.5% and 66.7% median) more CPU instructions than the two variants of EVALPERF, respectively. They also have 54% (25% median) more CPU instructions than TG-prompt. WEDGE tests dominate the number of programs (59%) where they run the

Technique	# of instructions ($\times 10^8$) Average	Median	Win rate	Slowdown over CC Average	Median
WEDGE	**5.96**	**0.75**	**59%**	**363**$\times$	**1.65**$\times$
TG-prompt	3.87 ($\downarrow$1.5$\times$)	0.60 ($\downarrow$1.3$\times$)	15%	275$\times$	1.52$\times$
EVALPERF$_{\text{SLOW}}$	3.23 ($\downarrow$1.8$\times$)	0.44 ($\downarrow$1.7$\times$)	12%	146$\times$	1.63$\times$
EVALPERF$_{\text{RAND}}$	3.21 ($\downarrow$1.9$\times$)	0.45 ($\downarrow$1.7$\times$)	14%	166$\times$	1.54$\times$

Table 5.1: WEDGE versus baselines described in §5.4.1. On average, our tool executes an average of about 84% and 85% (70% and 66% median), more CPU instructions than the two EVALPERF variants, respectively , and an average of about 54% (25% median) more CPU instructions than TG-prompt. WEDGE also outperforms all three baselines in a 4-way comparison of which technique slows down each tested program the most (denoted as "win rate"), by coming on top approximately 59% of the time. Finally, when compared with the original tests that the CodeContests dataset is released with, WEDGE outperforms them by nearly 240$\times$ on average (1.59$\times$ median).

slowest among all the other baselines (win rate).

Figure 5.3 visualizes the win rate by running head-to-head comparison between WEDGE and the baselines. Specifically, we count how many programs are slower when using our tests versus tests generated by the other techniques. We then divide this numbers in buckets of 10% size. A positive percentage indicates that PERFFORGE tests induce more slowdown, while a negative percentage indicates that the other technique in the head-to-head comparison does. So, higher bars in the right side of each graph indicate WEDGE outperforms that particular baseline.

WEDGE also outperforms the TG-prompt strategy, albeit by a smaller margin. The TG-prompt strategy is designed to synthesize an input generator by taking into account the problem specification and constraints, similar to [65, 101, 186]. We manually analyze a sample of these generators (§5.5). While the LLM attempts to implicitly reason about how to stress-test programs by inferring more diverse input patterns, these are ultimately focused on generic patterns specific to the broader class of algorithms that the problem implements (e.g., large graphs for graph-based problems). Consequently, there is little guarantee that these patterns reveal more complex performance bottlenecks.

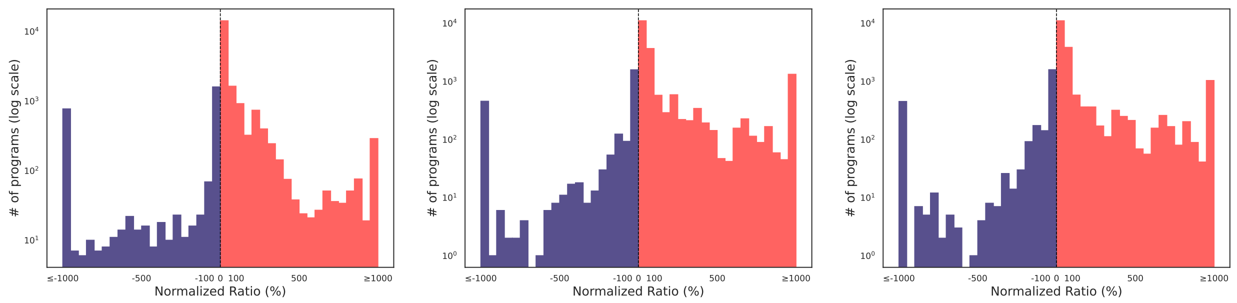
Technique	Execution time (ms)		Win rate
	Average	Median	
WEDGE	**228.60**	**82.33**	**59%**
TG-prompt	182.15 (↓1.3×)	85.60 (↑1.1×)	15%
EVALPERF$_{\text{SLOW}}$	139.83 (↓1.6×)	61.69 (↓1.3×)	12%
EVALPERF$_{\text{RAND}}$	134.79 (↓1.7×)	60.90 (↓1.4×)	14%

Table 5.2: WEDGE versus baselines described in §5.4.1. Similar to the results presented in Table 5.1, WEDGE outperforms all other techniques both in terms of physical running time and "win rate" (i.e., the 4-way comparison of which technique sloes down the most). The only exception is the median physical running time obtained by TG-prompt which is slightly slower than our tool. We attribute this anomaly to the fact that most of the C/C++ programs analyzed are I/O bound which is known to introduce moderate, but discernible amount of noise in physical time measurements [247].

Table Table 5.2 shows a similar trend when comparing each technique head-to-head while measuring physical running time. However, the measurement variance is more significant, reaching 12%. While multiple factors contribute to this large variance, we argue that the dominant factor of "noise" is I/O as a sizable fraction of programs need to read thousands of KB, thus making them I/O-bound.

We also compute the slowdown the tests achieved over the default tests in CodeContests. On average, WEDGE tests outperform EVALPERF ones by 2.3× and TG-prompt by 1.3×.

We extensively analyzed the performance-characterizing constraints as well as the test generators synthesized by WEDGE and the other baselines and benchmarks. We observe that the inputs generated by WEDGE focus more on the inefficient implementation in the code identified by the performance-characterizing constraints, while those by EVALPERF are optimized to stress the input length specified in the problem statement. TG-prompt, while not explicitly implemented to maximize bounds, faced challenges in reasoning about holistic program behaviors end-to-end. Even with chain-of-thought prompting, it still reduces to mostly generic length-stressing inputs specific to the problem statement (e.g., large graphs for graph-based problems). We leave the detailed description of these qualitative studies in §5.5 due to space constraints.



(a) Wedge vs. TG-prompt (b) Wedge vs. EvalPerf$_{\text{SLOW}}$ (c) Wedge vs. EvalPerf$_{\text{RAND}}$

Figure 5.3: A head-to-head comparison between PerfForge (🟥) and the baseline tests (🟪). The bars represent the number of programs where one incurs a larger number of CPU instructions. x-axis shows the corresponding ratio between the corresponding CPU instruction counts.

Table 5.3 shows the absolute numbers used to generate the histogram in Figure 5.3. Due to space constraints, we back the original 10% bins into 6 larger buckets. A bucket of `x .. y` represents the number of programs one technique incurs between x% and y% (inclusive) larger instruction count than the other. Specifically, the values outside of parentheses represent the number of programs that execute more instructions when running PerfForge tests, while those in parentheses indicate the number of programs that execute more instructions when running tests generated by one of the baselines (each row). Note that with one exception, TG-prompt tests that determine programs to execute 10× more instructions, our framework outperforms each technique. Even so, PerfForge tests determine more programs for longer by more than 50% compared to TG-prompt (Table 5.1).

5.4.3 Utility of PerfForge

As described in §5.4.1, we investigate the utility of PerfForge by comparing PerfForge tests to the default CodeContests tests (CC$_{\text{default}}$) that only evaluate the correctness on measuring performance improvement fairly by where the baseline's evaluation relied only on correctness tests generated by Pie [205] a state-of-the-art LLM-backed code optimization tool.

Techniques \ Bins	0 .. 199	200 .. 399	400 .. 599	600 .. 799	800 .. 999	1000+
WEDGE vs.						
TG-prompt	17,117 (↓1,704)	1,524 (↓63)	158 (↓60)	148 (↓40)	155 (↓27)	311 (↑768)
EVALPERF$_{\text{SLOW}}$	15,628 (↓1,856)	1,365 (↓69)	425 (↓43)	587 (↓ 7)	311 (↓11)	1,288 (↓454)
EVALPERF$_{\text{RAND}}$	15,879 (↓1,999)	966 (↓78)	588 (↓13)	664 (↓22)	367 (↓14)	1,004 (↓450)

Table 5.3: Source numbers for generating Figure 5.3. Due to space constraints, we group numbers in bins of size 200% instead of 10%. Amounts outside parenthesis indicate the number of programs where our technique slows down the execution more when compared head-to-head with one of the baselines techniques. In contrasts, amounts in parenthesis represent the number of program where one of the baseline techniques outperforms WEDGE by slowing down their execution more.

We show how PERFFORGE can measure performance improvement claimed by existing code optimization more fairly than the correctness test. To this end, we consider PIE [205], a state-of-the-art LLM-based code optimization based on finetuning, but relied on the default correctness tests to measure their performance improvement. We select their three most effective models (CodeLlama 13b) finetuned with the following different datasets: (1) HQ (high-quality) data annotated by the authors (PIE$_{\text{H}}$); (2) performance-conditioned data to optimize C++ programs annotated with a target optimization score reflecting its potential "peak performance" (PIE$_{\text{C}}$); (3) all data from the entire PIE dataset

We follow the same set of metrics as [205] by measuring the average relative speedup between the original and optimized code in instruction counts and physical time, as well as the percentage of programs that the LLM models can optimize by at least 10% (%Opt) [205]. §5.4.3 illustrates how our tests better characterize the performance bottlenecks. PERFFORGE outperforms the CodeContests default tests (CC$_{\text{default}}$) and its top five slowest tests (CC$_{\text{slow}}$) by 24% to 149% in terms of instruction counts and by 5% to 27% in terms of physical time. It also helps discover that between 7% and 48% more programs have actually been meaningfully optimized and run at least 10% faster.

Test set	Speedup (#inst)			Speedup (time)			%Opt (#inst)			%Opt (time)		
	PIE_H	PIE_C	PIE_A	PIE_H	PIE_C	PIE_A	PIE_H	PIE_C	PIE_A	PIE_H	PIE_C	PIE_A
$CC_{default}$	1.32	1.98	1.21	0.93	1.13	1.32	6.0%	16.2%	4.4%	16.6%	29.2%	56.5%
CC_{slow}	1.30	1.85	1.21	0.94	1.13	1.32	5.5%	15.6%	4.4%	18.1%	27.9%	56.5%
PerfForge	**1.62**	**1.99**	**3.01**	**1.18**	**1.29**	**1.38**	**12.1%**	**18.8%**	**30.4%**	**26.6%**	**41.6%**	**60.9%**

Table 5.4: Pie Experiment: average speedup and fraction of optimized programs, i.e., at least 10% faster (%Opt) evaluated by different test sets following [205]. The top-performing test set is highlighted.

5.4.4 *Sensitivity Analysis*

Comparison with plain fuzzing. We expand our investigation by comparing the performance of tests generated by WEDGE versus two sub-optimal versions of the tool: (1) guiding the mutator generation with constraints, but fuzzing the original (uninstrumented) program and (2) instrumenting the original program with constraint checker code, but using the default AFL++ mutators.

More detailed results presented in [236] shows that WEDGE's generated tests are on average between 50% and more than twice as slow in terms of CPU instruction count and relative slowdown 48.3% and 128.3% slower in case (1), and nearly 4× more CPU instructions than those generated.

Effect of input size. We investigate how input size affects the effectiveness of PERFFORGE considering that leveraging fuzzing to generate large inputs is a known challenging problem [137]. We observe our framework outperforms the baselines by larger margins when we further restrict the input size to be less than 1KB. In particular, for problems whose inputs are less than 1KB, the slowdown achieved by WEDGE is 3×, almost double that on the entire problems without such restrictions, 1.5×. These findings underscore that the performance-stressing characteristics of our tests stem from inputs being designed to target implementation-specific bottlenecks rather than being simply length-stressing.

To evaluate this, we partition our original dataset (§5.4.1) into subsets of problems based

Technique	# of instructions ($\times 10^8$) Average	Median	Win rate	Slowdown over CC Average	Median
		Less than 1MB			
WEDGE	**4.29**	**0.17**	**59%**	**113×**	**1.11×**
TG-prompt	2.28 (↓1.9×)	0.14 (↓1.2×)	11%	75×	1.01×
$\text{EVALPERF}_{\text{SLOW}}$	2.57 (↓1.7×)	0.07 (↓2.3×)	14%	57×	1.05×
$\text{EVALPERF}_{\text{RAND}}$	2.59 (↓1.7×)	0.08 (↓2.2×)	16%	88×	1.05×
		Less than 100KB			
WEDGE	**3.71**	**0.08**	**56%**	**88×**	**1.06×**
TG-prompt	1.85 (↓2.0×)	0.06 (↓1.3×)	10%	60×	1.00×
$\text{EVALPERF}_{\text{SLOW}}$	2.48 (↓1.5×)	0.03 (↓2.3×)	16%	49×	1.02×
$\text{EVALPERF}_{\text{RAND}}$	2.52 (↓1.5×)	0.03 (↓2.5×)	18%	84×	1.02×
		Less than 10KB			
WEDGE	**3.10**	**0.02**	**55%**	**20×**	**1.02×**
TG-prompt	1.26 (↓2.5×)	0.01 (↓1.5×)	5%	9×	1.00×
$\text{EVALPERF}_{\text{SLOW}}$	1.79 (↓1.7×)	0.01 (↓1.4×)	19%	7×	1.00×
$\text{EVALPERF}_{\text{RAND}}$	1.53 (↓2.0×)	0.01 (↓1.4×)	21%	8×	1.00×
		Less than 1KB			
WEDGE	**2.98**	**0.01**	**54%**	**5×**	**1.01×**
TG-prompt	1.00 (↓3.0×)	0.01 (↓1.5×)	6%	1×	1.00×
$\text{EVALPERF}_{\text{SLOW}}$	1.63 (↓1.8×)	0.01 (↓1.3×)	16%	1×	1.00×
$\text{EVALPERF}_{\text{RAND}}$	1.45 (↓2.1×)	0.01 (↓1.3×)	24%	1×	1.00×

Table 5.5: WEDGE versus baselines (§5.4.1) for problems whose inputs are below a threshold.

on the maximum input size they require, and compare PERFFORGE against tests by each baseline. Table 5.5 shows that on average, our framework performs better than its competitors when restricting input size. The largest shift happens when compared with TG-prompt. For problems whose inputs are less than 1KB, the difference is almost double than comparing the two on the entire dataset, 3.0× vs 1.5×, respectively. While we are still inspecting individual cases, but based on our preliminary investigation we hypothesize that this difference comes from the TG-prompt's focus on generating larger inputs with a bias towards universally edge-case patterns (e.g., near-complete graphs, sorted arrays).

5.5 Case Studies

In this section, we consider several interesting case studies of performance-characterizing constraints generated by WEDGE. We first rank the synthesized constraints by the combined size of their plain language description and corresponding generated checker code. We then select five out of the top ten problems that belong to different algorithmic classes. We make the full LLM answers publicly available along with PERFFORGE [4]

5.5.1 *Qualitative Analysis of Interesting Cases*

In this section, we investigate the test generators synthesized by each baseline technique and compare them with the performance-characterizing constraints generated by WEDGE, for Problem 633A which serves as our motivating example.

```python
def perf_input_gen(scale: int):
    # Assuming scale affects the complexity of the test, but since a and b have a fixed range,
    # we will use maximum values for a and b and scale will just determine c's complexity.
    a = 99
    b = 100
    c = 10_000  # Maximum possible value for c
    return (a, b, c)
```

Listing 5.1: EVALPERF$_{\text{SLOW}}$ synthesized generator for Problem 633A

Inspecting the test generator synthesized by EVALPERF$_{\text{SLOW}}$ (Listing 5.1) shows that it only generates inputs close to the problem's upper bounds (i.e., 99, 100, 10000) without even utilizing the `scale` parameter.

```python
def perf_input_gen(scale: int):
  # Set maximum values for a and b to test edge cases
  a = randint(1, 100)
  b = randint(1, 100)

  # Scale c to be large, but within the limit
  # c is chosen to be large to test performance, we still ensure it is within problem
      constraints
  c = min(max(1, scale), 10000)

  return (a, b, c)
```

Listing 5.2: EVALPERF$_{\text{RAND}}$ synthesized generator for Problem 633A

4. https://github.com/UChiSeclab/perfforge

The test generator synthesized EvalPerf$_{\text{RAND}}$ (Listing 5.2) random `a` and `b` within [1, 100] range and a large `c` (determined by `scale`) as the input. Still, the effect is generating values close to the upper bounds of the problem.

```
def generate_tests(output_directory):
  test_cases = [
    # Minimum edge case
    (1, 1, 1),

    # Simple cases
    (2, 3, 5),
    (3, 3, 6),

    # Co-prime (a, b) and larger c
    (17, 19, 100),
    (101, 103, 5000),

    # Large c with simple a, b
    (2, 3, 9999),

    # Boundary cases
    (50, 50, 10_000),
    (99, 100, 9_900),

    # Maximum values
    (100, 100, 10_000),

    # Various patterns
    (5, 10, 995),
    (7, 10, 9997),
    (1, 100, 10_000),
    (100, 1, 10_000),

    # Co-prime trickiness
    (4, 7, 9999),
    (15, 20, 10_000),

    # Just above/below multiples
    (3, 5, 10002),
    (5, 3, 9998),

    # Another complex co-prime situation
    (97, 89, 9999),

    # Randomized challenging cases
    (37, 41, 10000),
    (6, 9, 9999)
  ]
  # write test cases
  ...
```

Listing 5.3: TG-prompt synthesized generator for Problem 633A

TG-prompt (Listing 5.3) composes a series of concrete test cases rather than synthesizing a pattern-based generator. While it attempts to implicitly reason about how to synthesize performance-stressing inputs by including diverse patterns, some appear as generic corner cases. Thus, there is no guarantee these can trigger performance bottlenecks.

In contrast, Wedge relies on fuzzing with constraint-aware mutators to efficiently search for a diverse set of constraint-satisfying inputs. Listing 5.4 shows a constraint-aware mutator for Problem 633A. The mutator will mutate a previous input (seed input) to produce the next input (mutated input). The while loop will search the input space of (`a, b, c`) and ensure they conform to the constraints `not abs (a - b) > 5 or c % a == 0 or c % b == 0`, i.e., `abs (a - b) <= 5 and c % a != 0 and c % b != 0` (already satisfying the first two out of three constraints in Listing 5.5).

```python
def mutate_last_input(buf):
  """
  Mutate the last input slightly to explore the surrounding input space.
  """
  parts = buf.decode('utf-8').strip().split()
  a = int(parts[0])
  b = int(parts[1])
  c = int(parts[2])

  # Small mutations to each part
  a = max(1, min(100, a + random.randint(-5, 5)))
  b = max(1, min(100, b + random.randint(-5, 5)))
  c = max(1, c + random.randint(-500, 500))

  # Ensure the mutation leads to potentially challenging inputs
  while abs(a - b) > 5 or c % a == 0 or c % b == 0:
    a = max(1, min(100, a + random.randint(-5, 5)))
    b = max(1, min(100, b + random.randint(-5, 5)))
    c = max(1, c + random.randint(-500, 500))

  return f"{a} {b} {c}"
```

Listing 5.4: WEDGE synthesized constraint-aware fuzzing mutator for Problem 633A

```cpp
// Check if a and b are close and neither is a direct divisor of c
void check_close_values_constraint(int a, int b, int c) {
  if (abs(a - b) <= 5 && c % a != 0 && c % b != 0) {
    cerr << "Warning: close_values_constraint triggered - a and b are close, and neither
        divides c" << endl;
  }
}
```

Listing 5.5: WEDGE generated performance-characterizing constraints checker code

5.5.2 *Other Representative Case Studies*

# Case 1: Problem 1209B, solution #328. Given a set of n lights, each initially on/off and described by two parameters (a_i,b_i), light i toggles its state at times b_i, b_i+a_i, b_i+2a_i, ... The goal is to find the maximum number of lights simultaneously on at any moment.

```cpp
#include <bits/stdc++.h>
using namespace std;
const long double pie = 3.14159265358979;
const long long mod = 1e9 + 7;
string vow = "aeiou";
```

```cpp
 6 void solve(int test_case) {
 7   int n;
 8   cin >> n;
 9   string s;
10   cin >> s;
11   int N = 1e3;
12   vector<vector<int> > v(n, vector<int>(N, 0));
13   vector<pair<int, int> > p(n);
14   for (int i = 0; i < n; i++) cin >> p[i].first >> p[i].second;
15   for (int i = 0; i < n; i++) {
16     if (s[i] - '0') v[i][0] = 1;
17     for (int j = 1; j < p[i].second; j++) v[i][j] = v[i][j - 1];
18     int temp = v[i][0] ^ 1;
19     for (int j = p[i].second; j < N; j += p[i].first) {
20       for (int k = 0; j + k < N; k++) v[i][j + k] = temp;
21       temp ^= 1;
22     }
23   }
24   int ans = 0;
25   for (int j = 0; j < N; j++) {
26     int temp = 0;
27     for (int i = 0; i < n; i++) temp += v[i][j];
28     ans = max(ans, temp);
29   }
30   cout << ans;
31   cout << "\n";
32 }
33 int main() {
34   ios_base::sync_with_stdio(false);
35   cin.tie(0);
36   cout.tie(0);
37   int t = 1;
38   for (int i = 0; i < t; i++) solve(i);
39   return 0;
40 }
```

Listing 5.6: Example from our dataset: problem 1209B, solution 328

The code in Listing 5.6 simulates a fixed time window $N = 1,000$, instead of reasoning analytically about the periodic patterns which requires solving a least-common-multiple type of problem. Specifically, the program builds a two-dimensional array where for each light i it first copies its initial state up to $t = b_i$, then for each toggle epoch $t = b_i$,$b_i + a_i$,... it fills the remainder of the row in one shot via a nested "for k" loop at line 20. Finally, it scans each column t, sums up $v[i][t]$ for $i = 1, n$, and tracks the maximum (line 28). In the worst case, when $a_i = 1$, the program performs $O(N^2)$ iterations per toggling operation, for an overall complexity of $O(n \cdot N^2 + n \cdot N)$.

```cpp
void check_small_a_values(const vector<pair<int, int>>& p) {
    int small_a_count = 0;
    for (const auto& pair : p) {
        if (pair.first <= 2) {  // Assume 'small' a_i values are <= 2
            small_a_count++;
        }
    }
    if (small_a_count > 50) {  // Arbitrary threshold, adjust as needed
        cerr << "Warning: Performance bottleneck condition triggered - many lights have small 'a' values" << endl;
    }
}

void check_synchronized_b_values(const vector<pair<int, int>>& p) {
    map<int, int> b_count;
    for (const auto& pair : p) {
        b_count[pair.second]++;
    }
    for (const auto& [b_value, count] : b_count) {
        if (count > 30) {  // Arbitrary threshold for synchronization
            cerr << "Warning: Performance bottleneck condition triggered - synchronized 'b' values" << endl;
        }
    }
}

void check_large_number_of_lights(int n) {
    if (n > 90) {  // Close to the upper constraint
        cerr << "Warning: Performance bottleneck condition triggered - high number of lights" << endl;
    }
}
```

Listing 5.7: PC-constraints as C++ checker functions: problem 1209B, solution 328

WEDGE identifies three types of pc-constraints for this program, listed in 5.7.

1. The first performance-characterizing constraint (i.e., `check_small_a_values`) finds that if many lights have very small a_i (e.g. 1), translates to more iterations of the outer loop at line 15. Consequently, each iteration invokes a full inner copy (loop at line 20). Thus, small periods significantly amplify the work performed by the nested-loop work.

2. The second performance-characterizing constraint (`check_synchronized_b_values`) points to clusters of b_i values: if many lights share the same or close b_i, their current state lineup, causing the code to execute the heavy inner loop at line 20 for multiple lights at the same early offsets. Specifically, low or repeated `p[i].second` forces expensive fill operations to execute immediately and for many lights in fast succession.

3. Finally, the third perf-characterizing constraint (`check_large_number_of_lights`) simply indicates that as n approaches its upper bound near 100, the total nested work scales linearly in n. Each additional light multiplies the cost of the $O(N^2)$ toggling loops and the $O(N)$ scan across time.

A purely specification or problem-statement based performance analysis might determine that small toggle periods and a large number of lights trigger multiple loop iterations because any simulation of a periodic, large number of events would exhibit that. However, the actual performance profile of the code actual performance relies on two highly implementation-specific choices. First, instead of toggling cell by cell, the implementation writes an entire suffix of the time-array (for loop at line 20), thus a light toggle to linear ($O(N)$) instead of constant operation. Second, the choice of simulating the maximum number of seconds possible, $1,000\;1,000$, irrespective of the input. An optimal solution would take into account that each light's behavior is periodic, namely once it reaches its first toggle at $t = b_i$, thereafter it repeats every a_i seconds. This, naturally, translates into cycles of length equal to the *lowest common multiplier*: $\ell = \texttt{lcm}(a_1, a_2, \ldots, a_n)$. The complexity of the optimal program is, therefore, approximately two orders of magnitude since, based on the problem specifications, $1 \leq a_i, b_i \leq 5$. Since $\ell \leq lcm(1, 2, 3, 4, 5) = 60$ this leads to an optimal complexity of $O(\ell^2 \cdot n) = O(3,600 \cdot n)$ instead of the current $O(N^2 \cdot n) = O(10^6 \cdot n)$.

# Case 2: Problem 1118D1, solution #30. Given a homework of m pages and n coffee cups with caffeine doses a_1, ..., a_n, the problem asks to compute the minimum number of days a student must schedule to drink those cups so the pages he writes reach or exceed m. When the student drinks k cups on a single day labeled a_{i_1}, ..., a_{i_k}, the first cup lets him write a_{i_1} pages, the second max(0, $a_{i_2} - 1$), the third max(0, $a_{i_3} - 1$), and so on. Thus, the task is to find the smallest such sequence given n, m, and the list of caffeine values.

```
#include <bits/stdc++.h>
using namespace std;
const long long int N = 1e6;
long long int n, m;
vector<long long int> v(N);
bool check(long long int days) {
  long long int pages = 0, k = 0;
  for (long long int i = 0, j = 0; j < n; i++, j++) {
    pages += max(0ll, v[j] - k);
    if (i + 1 == days) i = -1, k++;
  }
  return pages >= m;
}
int32_t main() {
  scanf("%lld", &n);
  scanf("%lld", &m);
  long long int sum = 0;
  for (long long int i = 0; i < n; ++i) {
    scanf("%lld", &v[i]);
    sum += v[i];
21   }
22   if (sum < m) {
23     printf("-1");
24     return 0;
25   }
26   sort(v.begin(), v.end(), greater<long long int>());
27   long long int ans = 1e16;
28   long long int low = 1;
29   long long int high = n;
30   while (low <= high) {
31     long long int mid = (low + high) / 2;
32     if (check(mid)) {
33       ans = min(ans, mid);
34       high = mid - 1;
35     } else {
36       low = mid + 1;
37     }
38   }
39   printf("%lld", ans);
40   return 0;
41 }
```

Listing 5.8: Example from our dataset: problem 1118D1, solution 30

The program in Listing 5.8 implements a binary search on the expected output, m. For each value of m, the binary search simulates writing pages over *days* days (function `check`) by iterating each cup exactly once, decrementing future cups by an increasing offset k whenever a day's quota is reached at line (lines 9-10).

When the binary search checks a candidate value, it always scans all n cups in the for loop at line 8, performing $O(n)$ work per invocation. The binary search interval shrinks slowly, forcing multiple calls to check, which runs in $O(n)$ time. Thus, when n approaches its upper limit and m nearly equals the sum of all a_i, the combination of an $O(\log n)$ binary search multiplied by an linear check becomes the dominant bottleneck.

```
void check_binary_search_invariant(long long sum, long long m, int search_iterations) {
    if (search_iterations > 100 && sum >= m && (m > 0.9 * sum)) {
        cerr << "Warning: binary_search_invariant triggered - extended binary search due to close capacity and requirement" << endl;
        abort();
    }
}

void check_cup_order_invariant(const vector<long long>& v, long long m) {
    long long potential_pages = 0;
    int decrement_operations = 0;
    for (size_t i = 0; i < v.size(); ++i) {
        potential_pages += max(0ll, v[i] - (long long)i);
        if (v[i] > (long long)i) {
            ++decrement_operations;
        }
    }
    if (decrement_operations > 50 && potential_pages < m) {
        cerr << "Warning: cup_order_invariant triggered - extensive decrement operations" << endl;
    }
}
```

Listing 5.9: Performance-characterizing constraints as C++ checker functions: problem 1118D1, solution 30

Wedge identifies two types of performance-characterizing constraints for this program, listed in Listing 5.9.

1. The first performance-characterizing constraint (`check_binary_search_invariant`) finds that the binary search becomes expensive by performing a large number of iterations when the necessary pages m are close to the cumulative caffeine sum of all cups n and n is particularly large.

2. The second performance-characterizing constraint (`check_cup_order_invariant`) finds that the loop in `check` (lines 8-10) performs on the order of $n \times days$ operations. This happens because `check` iterates over the entire list of n cups adding `max`$(0, v[j] - k)$ per page and resetting the loop counter i every $days$ iterations.

The first performance-characterizing constraint relates to binary search complexity and is generic since searching over a large range is typically worst-case in terms of time complexity, the target value lies in the middle, forcing the search to "zig-zag" and perform a large number of iterations.

In contrast, the second performance-characterizing constraint is implementation-specific. The LLM reasons about how the check loop resets i and increments k to model diminishing returns and then calculates again $max(0, v[j] - k)$ for every cup. Note that the LLM borrows the same loop structure as in the original `check` function (e.g. reset of i and increment of k) so the checker code faithfully reproduces the slowdown pattern.

# Case 3: Problem 546C, solution #567. Given a deck of n distinct cards split arbitrarily into two decks, one per player), the problem asks to simulate a game in which, in each round, both players draw their top card, and the player with the higher value takes the opponent's card first and then their own, placing both at the bottom of their stack. When one player's stack becomes empty, the other wins.

```cpp
#include <bits/stdc++.h>
using namespace std;
queue<int> r1, r2;
int n, x, TLE, asd;
bool flag;
int main() {
  cin >> n >> x;
  for (int i = 1; i <= x; i++) cin >> asd, r1.push(asd);
  cin >> x;
  for (int i = 1; i <= x; i++) cin >> asd, r2.push(asd);
  while (TLE < 10000000) {
    if (r1.size() == 0 || r2.size() == 0) {
      flag = 1;
      break;
    }
    TLE++;
    int u = r1.front(), v = r2.front();
    r1.pop(), r2.pop();
    if (u > v)
      r1.push(v), r1.push(u);
    else
      r2.push(u), r2.push(v);
  }
  if (flag)
    if (r1.size() == 0)
      cout << TLE << " " << 2 << endl;
    else
      cout << TLE << " " << 1 << endl;
  else
    puts("-1");
  return 0;
}
```

Listing 5.10: Example from our dataset: problem 546C, solution 567

The program in Listing 5.10 simulates the game by representing the player as two queues. In each round, the implementation pops the head of each queue, compares the two values and adds both elements to the queue with the highest value of the two. If either queue is empty, the program terminates. While straightforward, the implementation has two significant flaws. First, the code does not store past game states after each round. This leads the simulation to repeat the same deck configurations repeatedly until the $TLE = 10^6$ threshold is met instead of breaking early if a cycle is detected, thus wasting many CPU cycles. Second, if the largest cards in either deck differ by a small amount (e.g., ≤ 1) the main loop will iterate for significantly more trials. When neither player's top card is consistently stronger, the pop/push operations keep both queues nearly the same size. This balanced exchange prevents either queue from becoming empty quick, forcing multiple extra iterations of the same pop/push logic.

```cpp
// Check if the queue configurations have repeated, indicating a cycle
void check_cycle_invariant(const queue<int>& r1, const queue<int>& r2, unordered_set<string>&
    seenConfigurations) {
    string config;
    auto copy1 = r1, copy2 = r2;
    while (copy1.empty()) config += tostring(copy1.front()) + ",";copy1.pop();config += "|";while
        (copy2.empty()) {
        config += to_string(copy2.front()) + ",";
        copy2.pop();
    }

    if (seenConfigurations.find(config) <> seenConfigurations.end()) {
        cerr << "Warning: Cycle detected - potential infinite loop" << endl;
    }
    seenConfigurations.insert(config);
}

// Check if neither player can win quickly due to balanced card strength
void check_balance_invariant(const queue<int>& r1, const queue<int>& r2) {
    int max1 = *max_element(r1.front(), r1.back());
    int max2 = *max_element(r2.front(), r2.back());
    if (abs(max1 - max2) <= 1) { // Arbitrarily chosen threshold for similar max strength
        cerr << "Warning: Balanced card strength - prolonged game possible" << endl;
    }
}

// Check for excessive number of rounds
void check_excessive_rounds(int TLE) {
    if (TLE > 1000) { // Example threshold, can be adjusted for practical purposes
        cerr << "Warning: Excessive number of game rounds" << endl;
    }
}
```

Listing 5.11: PC-constraints as C++ checker functions: problem 546C, solution 567

WEDGE identifies three types of performance-characterizing constraints for this program, listed in Listing 5.11.

1. The first performance-characterizing constraint (check_cycle_invariant) checks if the distribution of the two decks is prone to repeated states. The LLM detects that the original implementation does not take into account repetitions to terminate the main loop thus inputs with pathology is likely to force the loop at line 11 to iterate until reaching the $TLE = 10^6$ threshold wasting unnecessary cycles.

2. The second performance-characterizing constraint (check_balance_invariant) checks for "back-and-forth" push operation which cause minimal net change in queue size and prolong the game. The inefficiency comes from the two push calls per round (lines 20 and 22) and the fact that the losing card is enqueued first. This particular ordering choice leads to "reversing" of the first player's win since eventually, the smaller card,

which was enqueued first will move back to the second player, canceling the gains of the first one.

3. The third performance-characterizing constraint (`check_excessive_rounds`) simply reflect a generic and straightforward intuition: more loop iterations means more CPU cycles and more running time.

While the third performance-characterizing constraint is generic and could have been synthesized by an LLM without reasoning about the code, the first two are implementation-specific. Any queue-based simulation with a bounded state that can revisit prior configurations is prone to cycles. However, this is problematic only when such a program does not track repeated states, which is precisely what happens in this case. Moreover, the "back-and-forth" card exchange property is highly specific to the program the LLM is reasoning about. It happens precisely because of the choice of enqueuing order. Should the implementation enqueue the two values in the opposite order, it would be unlikely that there is any observable "back-and-forth" where cards move from one queue to another and then back again.

# Case 4: Problem 16B, solution #34. Given a set of m containers, where the i-th container holds a_i match boxes, each containing b_i matches, the goal is to select up to n boxes (without splitting boxes) to carry in a backpack so as to maximize the total number of matches carried away.

```cpp
#include <bits/stdc++.h>
using namespace std;
long long sumofdigits(string s) {
  long long sum = 0;
  for (long long i = 0; i < s.size(); i++) {
    int digit = s[i] - '0';
    sum += digit;
  }
  return sum;
}
int main() {
  int n;
  vector<pair<int, int>> v;
  cin >> n;
  int m;
  cin >> m;
  for (int i = 0; i < m; i++) {
```

```
18     int x, y;
19     cin >> x >> y;
20     pair<int, int> p(x, y);
21     v.push_back(p);
22   }
23   int sum = 0;
24   for (int i = 0; i < v.size() - 1; i++) {
25     for (int j = i + 1; j < v.size(); j++) {
26       if (v[j].second > v[i].second) {
27         pair<int, int> p = v[i];
28         v[i] = v[j];
29         v[j] = p;
30       }
31     }
32   }
33   int ans = 0;
34   for (int i = 0; i < v.size(); i++) {
35     int counter = 0;
36     if (sum == n) {
37       break;
38     }
39     int t = n - sum;
40     while (counter < v[i].first && t--) {
41       counter++;
42       sum++;
43       ans += v[i].second;
44     }
45   }
46   cout << ans << endl;
47   return 0;
48 }
```

Listing 5.12: Example from our dataset: problem 16B, solution 34

The code in Listing 5.12 performs two stages to solve the problem. First, it performs a simple $O(m^2)$ sort (lines 24-32) that orders the containers by their matches per box b_i descendingly. Second, it tries to fill the backpack in a greedy fashion (lines 34-45), as follows: For each container in sorted order, it enters a `while` loop (line 40) picks boxes one by one, decrementing the remaining capacity per box until either the container's supply is exhausted, or the capacity aggregator variable sum reaches n. Note that before entering the inner loop decrements t exactly once per box being packed (line 39). Thus a large n causes the code to run one iteration for each available box across all containers. Moreover, if n is much larger than the sum of all box capacities, the code still visits every box which yields a computational cost linear in the total capacity. Also, the code does not appear to terminate early if the box supply is exhausted. The code breaks out of the outer loop only if the current aggregated box capacity is equals to n (lines 36-37). If the supply is smaller, however, the outer loop iterates unnecessarily through all containers.

```
void check_large_n_invariant(int n, int m, const vector<pair<int, int>>& v) {
    long long totalBoxes = 0;
    for (const auto& container : v) {
        totalBoxes += container.first;
    }
    if (n > 10 * totalBoxes) {
        cerr << "Warning: Performance bottleneck condition triggered - n is much larger than
            available matchboxes" << endl;
    }
}

void check_small_total_boxes_invariant(int n, int m, const vector<pair<int, int>>& v) {
    long long totalBoxes = 0;
    for (const auto& container : v) {
        totalBoxes += container.first;
    }
    if (totalBoxes < n / 10) {
        cerr << "Warning: Performance bottleneck condition triggered - insufficient matchboxes
            compared to n" << endl;
    }
```

Listing 5.13: PC-constraints as C++ checker functions: problem 16B, solution 34

WEDGE identifies two types of pc-constraints for this program, listed in 5.7.

The first performance-characterizing constraint (`check_large_n_invariant`) detects when the target capacity n vastly exceeds the total number of available boxes. The inner loop (lines 40-44) executes exactly one iteration per box taken. So, the mode infers that when n is significantly larger than the available boxes, the code performs $O(\sum_i a_i)$ iterations since each time it decrements t by 1.

The second perf-characterizing constraint (`check_small_total_boxes_invariant`) detects when the aggregate supply of boxes is substantially less than n, since the loop break condition (line 37) applies only when the bag is exactly full. Otherwise, the outer loop (line 34) iterates over all containers, wasting cycles when capacity is no longer available.

A purely specification or problem-statement based performance analysis could infer that larger n could cause longer execution, and that a typical implementation would iterate until the capacity is full or exceeded. However, the first constraint hinges on the implementation choice of simulating each box by decrementing t. This is suboptimal because to solve the problem, the algorithm only needs to know how many boxes to take, not to process them individually. Similarly, the second constraint speculates that the code does not exit immediately once it is determined that the matchbox supply is depleted before reaching n.

5.6 Summary

This chapter reframes performance stress test generation as a two-step search problem: (i) reasoning about *performance-characterizing constraints*—predicates on program state that indicate worst-case performance behavior—and (ii) searching for inputs that satisfy them.

Guided by a contrast between fast and slow execution profiles, WEDGE identifies cost-amplifying (hot) code regions and synthesizes Boolean checkers in the form of ordinary branches that existing coverage engines could consume unchanged. A constraint-aware mutator then steers the fuzzer toward grammatically valid, constraint-satisfying inputs, continually pushing the workload further into worst-case territory and exposing complex inefficiencies invisible to correctness-centric tests.

We showed that WEDGE can outperform all baseline techniques on 59% of the programs tested, on average slowing down execution 1.5× more than the second-best baseline. We further demonstrated that PERFFORGE helps better evaluate and improve existing code optimization techniques, such as PIE [205]. We released our performance-stressing tests, along with the CodeContest programs, as a new benchmark PERFFORGE [5].

The methodology developed in this chapter combines the advantage of LLMs' reasoning abilities, namely *fuzzy their code comprehension capabilities*, with the guided searching ability of fuzzing. We hope our work marks a step toward fully automated, domain agnostic performance analysis.

5. https://github.com/UChiSeclab/perfforge

CHAPTER 6

CONCLUSION AND FUTURE WORK

This dissertation demonstrated that latent, complex bugs present in modern software systems can be systematically exposed by perturbing ordinary (test) executions with lightweight, semantics-aware strategies. This work described three systems—WAFFLE, WASABI, and WEDGE—showing that forcing small, semantics-guided disruptions while running existing tests, reliably expose memory ordering violations (MEMORDER bugs), faulty retry logic, and obscure performance bottlenecks that traditional analysis tools and testing frameworks miss. Each system operates at a increasingly broader semantic scope: WAFFLE delays individual statements to force adversarial thread interleavings, WASABI injects transient faults to steer retry mechanisms into rarely-exercised faulty states, and WEDGE synthesizes branch-level performance-characterizing constraints to guide fuzzing toward exposing code inefficiencies. In every case, execution perturbation is informed by lightweight program analysis often complemented by AI-guided code comprehension techniques, in order to strike a balance between scalability, efficiency, and bug detection coverage.

While the research presented in this dissertation attempts to cover many of the existing literature gaps when it comes to surfacing latent, complex bugs in modern systems, important open problems remain for future work.

Resiliency functionality on emerging platforms. Traditional resilience logic (task retry, throttling, cancellation) assumes failures are rare and transient. These assumptions, however, do not always hold on emerging computing platforms such as cloud native systems. Our previous work (Chapter 4) studies faulty retry patterns and discovered that some, like those triggered during redeployment, can overwhelm resources and mask real incidents [211].

Cloud native systems amplify volatility through frequent container restarts, IP reassignments, and auto-scaling. What once appeared as a failure can be a routine scale-down or redeployment. Oversimplified retry policies, especially under such cursory conditions, risk overwhelming the system and obscuring faults. Although back-off and circuit breakers mitigate transient errors, they often cannot distinguish planned reconfigurations from real disruptions.

Thus, traditional resilience paradigms need to be revisited in the context of these emerging platforms. One potential solution is developing context-aware resilience logic based on telemetry, to make that distinction. Consequently, future work could investigate how resilience functionality can evolve to integrate cluster-level insights (e.g., rollout updates) and become platform-level functionality, rather than being retrofitted after-the-fact.

Understanding and addressing system unpredictability at scale. Recent studies on cloud-scale incidents [80, 93, 105, 135, 212, 217] reveal that modern large-scale systems are often tested in well-provisioned environments that mask performance issues, yet real deployments frequently run at capacity due to cost or workload spikes. When memory, CPU, or I/O saturate, or software constraints like connection limits are reached, operational issues can increase unpredictably and disrupt normal operation. Operators and software engineers often fail to reproduce these issues in smaller-scale test environments, making capacity-driven failures a major cause of high-impact incidents [80, 93, 105]. Understanding how capacity constraints affect large-scale performance is crucial for closing the gap between "works as expected when deployed on 10 nodes" and "fails when scaled to 100 nodes".

To address this challenge, future work could continue investigating how capacity saturation leads to operational unpredictability. Being "at capacity" goes beyond CPU or memory "being full". It can also mean that the system reached throughput/latency boundaries or internal operational limits (e.g., maximum connections, data structure load, etc.), often obscured from operators and engineers. While existing tools excel at identifying the former, visible or "external" pressure points, they often struggle to pinpoint the latter, less visible or "internal" ones, thus falling short of isolating likely performance bottlenecks.

AI-informed incident flow orchestration. Modern large-scale systems produce vast streams of operational data that demand effective, informed incident responses. Currently available solutions often piece together a variety of monitoring tools, log analyzers, and documentation mining, yet do not seamlessly integrate historical knowledge and real-time context. This makes automated cross-referencing and contextual linking more challenging, and prevent the incident response layer (and, ultimately, system operators) to adapt to new and emerging failure patterns.

Future work could therefore explore designing an "incident orchestration" layer that integrates AI approaches with existing observability to coordinate response from alert to post-mortem analysis. Once an incident is detected, the system retrieves relevant logs, traces, metrics, and recent change events using structure-aware search with semantic matching, retaining provenance. This information could then populates a typed incident graph whose nodes represent services, software versions, resources, symptoms, and candidate causes, and whose edges capture temporal and causal relations between them. By matching this graph against prior incidents, the incident orchestration layer assigns labels and priority, and proposes ranked mitigation strategies with explicit preconditions and expected effects. For example, recent research on production cloud incidents [80, 105, 155, 217] showed that software engineers submit short, straightforward mitigation fixes (e.g., rolling back to an older version, restarting nodes or processes) in over 50% of incidents reported. These actions can be easily automated with the appropriate safety checks through policy and dependency validation, sandboxing, and staged rollout, as well as going through operator approval for more complex systemic changes.

AI support for system design. Modern systems are developed and deployed at a pace that assumes software engineers can quickly perform baseline evaluations. In other words, obtain source code, resolve dependencies, configure environments, execute test cases, and verify specifications. In practice, documentation is incomplete and execution environments diverge, turning setup and validation into a time-consuming task. These gaps can also mask regressions and undermine reproducibility, portability, and reusability.

A promising direction is an AI-assisted framework that reads code repositories and documentation to construct deployment plans and scripts, infer prerequisites, and run minimal, representative usecases that check documented properties. Such an agent would package configurations in containers, verify consistency between containerized and host versions, report mismatches, and suggests actionable steps, code patches, or checklists for software engineers to review. Future work should examine how such agents integrate with CI/CD and change-management systems so that reproducibility becomes a routine automated function rather than a manual task or, worse, an afterthought. Beyond deployment, the same agents can parse specifications and design documents, align design assumptions with the implementation, flag omissions or deviations, proposing patches along with optimizations under various performance and correctness constraints.

This direction is, arguably, the logical next step beyond AI-guided approaches for software engineering.

REFERENCES

[1] ApplicationInsights.NET Issue # 1106. Retrieved October 4, 2022. (Retrieved October 4, 2022). https://github.com/microsoft/ApplicationInsights-dotnet/issues/1106.

[2] NetMQ Issue # 814. Retrieved October 4, 2022. (Retrieved October 4, 2022). https://github.com/zeromq/netmq/issues/814.

[3] NetMQ Issue # 975. Retrieved October 4, 2022. (Retrieved October 4, 2022). https://github.com/zeromq/netmq/issues/975.

[4] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 74–89. https://doi.org/10.1145/945445.945454

[5] Adil Ahmad, Sangho Lee, Pedro Fonseca, and Byoungyoung Lee. 2021. Kard: lightweight data race detection with per-thread memory protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 647–660. https://doi.org/10.1145/3445814.3446727

[6] Toufique Ahmed and Premkumar Devanbu. 2023. Few-shot training LLMs for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (, Rochester, MI, USA,) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 177, 5 pages. https://doi.org/10.1145/3551349.3559555

[7] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl T. Barr. 2024. Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization). In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE) (ICSE'24)*. IEEE Computer Society, Los Alamitos, CA, USA, 1004–1004. https://doi.ieeecomputersociety.org/

[8] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Correlated crash vulnerabilities. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 151–167.

[9] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. 2018. An analysis of network-partitioning failures in cloud systems. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 51–68.

[10] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. 2015. Lineage-driven Fault Injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 331–346. https://doi.org/10.1145/2723372.2723711

[11] Anastasios Andronidis and Cristian Cadar. 2022. SnapFuzz: high-throughput fuzzing of network applications. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 340–351. https://doi.org/10.1145/3533767.3534376

[12] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. https://doi.org/10.14722/ndss.2019.23412

[13] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. 2022. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868* (2022).

[14] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 748–758. https://doi.org/10.1109/ICSE.2019.00083

[15] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[16] Amazon AWS. [n. d.]. Summary of the Amazon Kinesis Data Streams service event in Northern Virginia (US-EAST-1) region. ([n. d.]). https://aws.amazon.com/message/073024/

[17] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. FUDGE: fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 975–985. https://doi.org/10.1145/3338906.3340456

[18] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. 2019. Effective Static Analysis of Concurrency Use-after-Free Bugs in Linux Device Drivers. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 255–268.

[19] Radu Banabic and George Candea. 2012. Fast black-box testing of system recovery code. In *Proceedings of the 7th ACM European Conference on Computer Systems*

(Bern, Switzerland) *(EuroSys'12)*. Association for Computing Machinery, New York, NY, USA, 281–294. https://doi.org/10.1145/2168836.2168865

[20] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using Magpie for Request Extraction and Workload Modelling. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/osdi-04/using-magpie-request-extraction-and-workload-modelling

[21] Nils Bars, Moritz Schloegel, Nico Schiller, Lukas Bernhard, and Thorsten Holz. 2024. No Peer, no Cry: Network Application Fuzzing via Fault Injection. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 750–764. https://doi.org/10.1145/3658644.3690274

[22] Eric Berger. [n. d.]. Starliner faced "catastrophic" failure before software bug found. *ArsTechnica* ([n. d.]). https://arstechnica.com/science/2020/02/starliner-faced-catastrophic-failure-before-software-bug-found/

[23] Emery D Berger. 2020. Scalene: Scripting-language aware profiling for python. *arXiv preprint arXiv:2006.03879* (2020).

[24] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[25] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: Compositional Static Race Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 144 (oct 2018), 28 pages. https://doi.org/10.1145/3276514

[26] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2329–2344.

[27] Arkady Bron, Eitan Farchi, Yonit Magid, Yarden Nir, and Shmuel Ur. 2005. Applications of Synchronization Coverage. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Chicago, IL, USA) *(PPoPP '05)*. Association for Computing Machinery, New York, NY, USA, 206–212. https://doi.org/10.1145/1065944.1065972

[28] Nathan Bronson, Abutalib Aghayev, Aleksey Charapko, and Timothy Zhu. 2021. Metastable failures in distributed systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 221–227. https://doi.org/10.1145/3458336.3465286

[29] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. *SIGPLAN Not.* 45, 3 (March 2010), 167–178. https://doi.org/10.1145/1735971.1736040

[30] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. 2009. WISE: Automated test generation for worst-case complexity. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, USA, 463–473. https://doi.org/10.1109/ICSE.2009.5070545

[31] Juan Caballero, Gustavo Grieco , Mark Marron, and Antonio Nappa . 2012. Undangle: Early Detection of Dangling Pointers in Use-After-Free and Double-Free Vulnerabilities. In *ISSTA 2012 Proceedings of the 2012 International Symposium on Software Testing and Analysis* (issta 2012 proceedings of the 2012 international symposium on software testing and analysis ed.). Association for Computing Machinery, 133–143. https://www.microsoft.com/en-us/research/publication/undangle-early-detection-dangling-pointers-use-free-double-free-vulnerabilities/

[32] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[33] Yan Cai, Biyun Zhu, Ruijie Meng, Hao Yun, Liang He, Purui Su, and Bin Liang. 2019. Detecting Concurrency Memory Corruption Vulnerabilities. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 706–717. https://doi.org/10.1145/3338906.3338927

[34] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. 2012. A NICE way to test openflow applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA) *(NSDI'12)*. USENIX Association, USA, 10.

[35] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227* (2022).

[36] Bihuan Chen, Yang Liu, and Wei Le. 2016. Generating performance distributions via probabilistic symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/2884781.2884794

[37] Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. 2021. CoFI: consistency-guided fault injection for cloud systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia)

(ASE'20). Association for Computing Machinery, New York, NY, USA, 536–547. https://doi.org/10.1145/3324884.3416548

[38] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[39] Xu Chen, Ming Zhang, Z. Morley Mao, and Paramvir Bahl. 2008. Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 117–130.

[40] Yinfang Chen, Xudong Sun, Suman Nath, Ze Yang, and Tianyin Xu. 2023. Push-Button Reliability Testing for Cloud-Backed Applications with Rainmaker. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023 (Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023)*. USENIX Association, USA, 1701–1716.

[41] Ankit Choudhary, Shan Lu, and Michael Pradel. 2017. Efficient Detection of Thread Safety Violations via Coverage-Guided Generation of Concurrent Tests. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 266–277. https://doi.org/10.1109/ICSE.2017.32

[42] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. 2014. The Mystery Machine: End-to-End Performance Analysis of Large-Scale Internet Services. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 217–231.

[43] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. 2014. The mystery machine: end-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 217–231.

[44] Maria Christakis, Patrick Emmisberger, Patrice Godefroid, and Peter Müller. 2017. A general framework for dynamic stub injection. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE'17)*. IEEE Press, 586–596. https://doi.org/10.1109/ICSE.2017.60

[45] Patrick Cousot. 1996. Abstract interpretation. *ACM Computing Surveys (CSUR)* 28, 2 (1996), 324–328.

[46] Lucy Craymer. [n. d.]. 'Leap year glitch' shuts some New Zealand fuel pumps. *Reuters* ([n. d.]). https://www.reuters.com/world/asia-pacific/leap-year-glitch-shuts-some-new-zealand-fuel-pumps-2024-02-29/

[47] Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. 2024. Meta large language model compiler: Foundation models of compiler optimization. *arXiv preprint arXiv:2407.02524* (2024).

[48] Charlie Curtsinger and Emery D. Berger. 2018. Coz: finding code that counts with causal profiling. *Commun. ACM* 61, 6 (May 2018), 91–99. https://doi.org/10.1145/3205911

[49] Arnaldo Carvalho De Melo. 2010. The new linux'perf'tools. In *Slides from Linux Kongress*, Vol. 18. 1–42.

[50] Chris DeBrusk. [n. d.]. What Should We Do to Prevent Software From Failing? *The MIT Sloan Management Review* ([n. d.]). https://sloanreview.mit.edu/article/what-should-we-do-to-prevent-software-from-failing/

[51] DeepMind. 2022. CodeContests Dataset. https://huggingface.co/datasets/deepmind/code_contests. Codeforces section, problem 633_A. Ebony and Ivory..

[52] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 127–144. https://doi.org/10.1145/2541940.2541941

[53] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA'23)*. Association for Computing Machinery, New York, NY, USA, 423–435. https://doi.org/10.1145/3597926.3598067

[54] Yangruibo Ding, Benjamin Steenhoek, Kexin Pei, Gail Kaiser, Wei Le, and Baishakhi Ray. 2024. TRACED: Execution-aware Pre-training for Source Code. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering* (, Lisbon, Portugal,) *(ICSE'24)*. Association for Computing Machinery, New York, NY, USA, Article 36, 12 pages. https://doi.org/10.1145/3597503.3608140

[55] Apache Cassandra Docs. Accessed: April 2024. https://cassandra.apache.org/doc/stable/cassandra/configuration/cass_yaml_file.html.

[56] Apache Cassandra Docs. Accessed: April 2024. https://www.elastic.co/guide/en/elasticsearch/hadoop/8.13/configuration.html.

[57] Apache HDFS Docs. Accessed: April 2024. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml.

[58] Apache Hive Docs. Accessed: April 2024. https://cwiki.apache.org/confluence/display/Hive/Configuration+Properties.

[59] Apache HBase Docs. Accessed: April 2024. https://hbase.apache.org/book.html.

[60] Apache MapReduce Docs. Accessed: April 2024. https://hadoop.apache.org/docs/r3.1.0/hadoop-project-dist/hadoop-common/core-default.xml.

[61] Apache MapReduce Docs. Accessed: April 2024. https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/mapred-default.xml.

[62] Apache Yarn Docs. Accessed: April 2024. https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/yarn-service/Configurations.html.

[63] CodeQL Documentation. Accessed: April 2024. https://codeql.github.com/docs/.

[64] Polly documentation. Accessed: April 2024. https://www.pollydocs.org.

[65] Mingzhe Du, Luu Anh Tuan, Bin Ji, Qian Liu, and See-Kiong Ng. 2024. Mercury: A Code Efficiency Benchmark for Code Large Language Models. *Advances in Neural Information Processing Systems* 37 (2024).

[66] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. 2007. Blended analysis for performance understanding of framework-based applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis* (London, United Kingdom) *(ISSTA '07)*. Association for Computing Machinery, New York, NY, USA, 118–128. https://doi.org/10.1145/1273463.1273480

[67] Ian Duncan, Daniel Gilbert, Lori Aratani, Shira Ovide, and Danny Nguyen. [n. d.]. Microsoft, CrowdStrike outage disrupts travel and business worldwide. *The Washington Post* ([n. d.]). https://www.washingtonpost.com/business/2024/07/19/windows-outage-crowdstrike-cancellations-computer/

[68] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2007. Goldilocks: a race and transaction-aware java runtime. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 245–255. https://doi.org/10.1145/1250734.1250762

[69] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 237–252. https://doi.org/10.1145/945445.945468

[70] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective Data-Race Detection for the Kernel. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 151–162.

[71] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69, 1–3, 35–45. https://doi.org/10.1016/j.scico.2007.01.015

[72] Jb Evain. Retrieved October 4, 2022. Mono.Cecil. (Retrieved October 4, 2022). https://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil.

[73] Jack Ewing. [n. d.]. As Automakers Add Technology to Cars, Software Bugs Follow. *The New York Times* ([n. d.]). https://www.nytimes.com/2022/02/08/business/car-software-lawsuits.html

[74] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[75] Emily First, Markus Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-Proof Generation and Repair with Large Language Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (, San Francisco, CA, USA,) *(ESEC/FSE'23)*. Association for Computing Machinery, New York, NY, USA, 1229–1241. https://doi.org/10.1145/3611643.3616243

[76] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. *SIGPLAN Not.* 44, 6, 121–133. https://doi.org/10.1145/1543135.1542490

[77] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. 2014. SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 415–431.

[78] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[79] Sean Gallagher. May 2017. WCry ransomware worm's Bitcoin take tops $70k as its spread continues. https://arstechnica.com/information-technology/2017/05/wcry-ransomware-worms-bitcoin-take-tops-70k-as-its-spread-continues/.

[80] Supriyo Ghosh, Manish Shetty, Chetan Bansal, and Suman Nath. 2022. How to fight production incidents? an empirical study on a large-scale cloud service. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 126–141. https://doi.org/10.1145/3542929.3563482

[81] Patrice Godefroid. 1997. Model Checking for Programming Languages Using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France) *(POPL '97)*. Association for Computing Machinery, New York, NY, USA, 174–186. https://doi.org/10.1145/263699.263717

[82] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. 2021. Snowboard: Finding Kernel Concurrency Bugs through Systematic Inter-Thread Communication Analysis. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 66–83. https://doi.org/10.1145/3477132.3483549

[83] Sishuai Gong, Dinglan Peng, Deniz Altınbüken, Pedro Fonseca, and Petros Maniatis. 2023. Snowcat: Efficient Kernel Concurrency Testing using a Learned Coverage Predictor. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 35–51. https://doi.org/10.1145/3600006.3613148

[84] Google. Retrieved July 1, 2025. Jackalope. https://github.com/googleprojectzero/Jackalope.

[85] Google. Retrieved July 1, 2025. Syzkaller. https://github.com/google/syzkaller.

[86] Harrison Green and Thanassis Avgerinos. 2022. GraphFuzz: Library API Fuzzing with Lifetime-aware Dataflow Graphs. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 1070–1081. https://doi.org/10.1145/3510003.3510228

[87] Jiawei Tyler Gu, Xudong Sun, Wentao Zhang, Yuxuan Jiang, Chen Wang, Mandana Vaziri, Owolabi Legunsen, and Tianyin Xu. 2023. Acto: Automatic End-to-End Testing for Operation Correctness of Cloud System Management. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 96–112. https://doi.org/10.1145/3600006.3613161

[88] Binfa Gui, Wei Song, and Jeff Huang. 2021. UAFSan: An Object-Identifier-Based Dynamic Approach for Detecting Use-after-Free Vulnerabilities. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 309–321. https://doi.org/10.1145/3460319.3464835

[89] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. 2011. FATE and DESTINI: a framework for cloud recovery testing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA) *(NSDI'11)*. USENIX Association, USA, 238–252.

[90] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/2670979.2670986

[91] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. 2016. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SoCC'16)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/2987550.2987583

[92] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dussea, and Ben Liblit. 2008. EIO: error handling is occasionally correct *(FAST'08)*. USENIX Association, USA, Article 14, 16 pages.

[93] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. 2018. Fail-slow at scale: evidence of hardware performance faults in large production systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies* (Oakland, CA, USA) *(FAST'18)*. USENIX Association, USA, 1–14.

[94] Mark Harman. 2007. The Current State and Future of Search Based Software Engineering. In *Future of Software Engineering (FOSE '07)*. 342–357. https://doi.org/10.1109/FOSE.2007.29

[95] Nikolas Havrikov and Andreas Zeller. 2019. Systematically Covering Input Structure. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 189–199. https://doi.org/10.1109/ASE.2019.00027

[96] Jingzhu He, Yuhang Lin, Xiaohui Gu, Chin-Chia Michael Yeh, and Zhongfang Zhuang. 2022. PerfSig: extracting performance bug signatures via multi-modality causal analysis. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1669–1680. https://doi.org/10.1145/3510003.3510110

[97] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*. https://openreview.net/forum?id=sD93GOzH3i5

[98] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K. Reiter, and Vyas Sekar. 2016. Gremlin: Systematic Resilience Testing of Microservices. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. 57–66. https://doi.org/10.1109/ICDCS.2016.11

[99] Egor Homakov. May 2015. Hacking Starbucks for unlimited coffee. (May 2015). https://www.dailydot.com/unclick/starbucks-hack-unlimited-coffee-2015/.

[100] Shin Hong, Jaemin Ahn, Sangmin Park, Moonzoo Kim, and Mary Jean Harrold. 2012. Testing Concurrent Programs to Achieve High Synchronization Coverage. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (Minneapolis, MN, USA) *(ISSTA 2012)*. Association for Computing Machinery, New York, NY, USA, 210–220. https://doi.org/10.1145/2338965.2336779

[101] Dong HUANG, Jianbo Dai, Han Weng, Puzhen Wu, Yuhao QING, Heming Cui, Zhijiang Guo, and Jie Zhang. 2024. EffiLearner: Enhancing Efficiency of Generated Code via Self-Optimization. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=KhwOuB0fs9

[102] Dong HUANG, Yuhao QING, Weiyi Shang, Heming Cui, and Jie Zhang. 2024. EffiBench: Benchmarking the Efficiency of Automatically Generated Code. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. https://openreview.net/forum?id=30XanJanJP

[103] Dong Huang, Guangtao Zeng, Jianbo Dai, Meng Luo, Han Weng, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie M Zhang. 2024. Effi-code: Unleashing code efficiency in language models. *arXiv preprint arXiv:2410.10209* (2024).

[104] Jeff Huang. 2018. UFO: Predictive Concurrency Use-after-Free Detection. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 609–619. https://doi.org/10.1145/3180155.3180225

[105] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. 2022. Metastable Failures in the Wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22) (OSDI'22)*. USENIX Association, Carlsbad, CA, 73–90. https://www.usenix.org/conference/osdi22/presentation/huang-lexiang

[106] Netflix Hystrix. Accessed: April 2024. https://github.com/Netflix/Hystrix.

[107] LLM Compiler Infrastructure. Retrieved July 1, 2025. LibFuzzer. https://llvm.org/docs/LibFuzzer.htm.

[108] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua

Wang. 2018. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 519–532. https://www.usenix.org/conference/atc18/presentation/iorgulescu

[109] Joab Jackson. May 2012. Nasdaq's Facebook glitch came from 'race conditions'. https://www.computerworld.com/article/2504676/nasdaq-s-facebook-glitch-came-from--race-conditions-.html.

[110] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE'22)*. Association for Computing Machinery, New York, NY, USA, 1219–1231. https://doi.org/10.1145/3510003.3510203

[111] Suman Jana, Yuan Kang, Samuel Roth, and Baishakhi Ray. 2016. Automatically detecting error handling bugs using error specifications. In *Proceedings of the 25th USENIX Conference on Security Symposium* (Austin, TX, USA) *(SEC'16)*. USENIX Association, USA, 345–362.

[112] Dae R. Jeong, Yewon Choi, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. 2024. OZZ: Identifying Kernel Out-of-Order Concurrency Bugs with In-Vivo Memory Access Reordering. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 229–248. https://doi.org/10.1145/3694715.3695944

[113] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding Kernel Race Bugs through Fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*. 754–768. https://doi.org/10.1109/SP.2019.00017

[114] Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023. LLMLingua: Compressing Prompts for Accelerated Inference of Large Language Models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 13358–13376. https://doi.org/10.18653/v1/2023.emnlp-main.825

[115] Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2024. LongLLMLingua: Accelerating and Enhancing LLMs in Long Context Scenarios via Prompt Compression. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 1658–1677. https://aclanthology.org/2024.acl-long.91

[116] Zongze Jiang, Ming Wen, Jialun Cao, Xuanhua Shi, and Hai Jin. 2024. Towards Understanding the Effectiveness of Large Language Models on Directed Test Input Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1408–1420. https://doi.org/10.1145/3691620.3695513

[117] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/2254064.2254075

[118] Xiaoen Ju, Livio Soares, Kang G. Shin, Kyung Dong Ryu, and Dilma Da Silva. 2013. On fault resilience of OpenStack. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (Santa Clara, California) *(SOCC'13)*. Association for Computing Machinery, New York, NY, USA, Article 2, 16 pages. https://doi.org/10.1145/2523616.2523622

[119] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. 2022. TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3219–3236. https://www.usenix.org/conference/usenixsecurity22/presentation/kande

[120] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) (ICSE'23)*. 2312–2323. https://doi.org/10.1109/ICSE48619.2023.00194

[121] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. 2017. Lazy Diagnosis of In-Production Concurrency Bugs. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 582–598. https://doi.org/10.1145/3132747.3132767

[122] Baris Kasikci, Cristian Zamfir, and George Candea. 2012. Data Races vs. Data Race Bugs: Telling the Difference with Portend. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) *(ASPLOS XVII)*. Association for Computing Machinery, New York, NY, USA, 185–198. https://doi.org/10.1145/2150976.2150997

[123] Baris Kasikci, Cristian Zamfir, and George Candea. 2013. RaceMob: Crowdsourced Data Race Detection. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for

Computing Machinery, New York, NY, USA, 406–422. https://doi.org/10.1145/2517349.2522736

[124] Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. 2017. *Data flow analysis: theory and practice*. CRC Press.

[125] Jiwon Kim, Benjamin E. Ujcich, and Dave (Jing) Tian. 2023. Intender: Fuzzing Intent-Based Networking with Intent-State Transition Guidance. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 4463–4480. https://www.usenix.org/conference/usenixsecurity23/presentation/kim-jiwon

[126] Kyle Kingsbury and Peter Alvaro. 2020. Elle: inferring isolation anomalies from experimental observations. *Proc. VLDB Endow.* 14, 3 (nov 2020), 268–280. https://doi.org/10.14778/3430915.3430918

[127] Ryan Knutson and Jessica Mendoza. [n. d.]. The Botched Software Update That Cost 600 Million US Dollars. *The Wall Street Journal* ([n. d.]). https://www.wsj.com/podcasts/the-journal/the-botched-software-update-that-cost-600-million/98bede46-a380-4fbf-bee0-f320f1a19d90

[128] Xinghui Kok and Archishma Iyer. [n. d.]. Singapore's central bank tells DBS, Citibank to investigate system outage. *Reuters* ([n. d.]). https://www.reuters.com/business/finance/singapores-central-bank-tells-dbs-citibank-investigate-system-outage-2023-10-19/

[129] Herb Krasner. [n. d.]. The Cost of Poor Sofware Quality in the US: A 2022 Report. *Consortium for Information and Software Quality* ([n. d.]). https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2022-report

[130] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (jul 1978), 558–565. https://doi.org/10.1145/359545.359563

[131] Hayley LeBlanc, Shankara Pailoor, Om Saran K R E, Isil Dillig, James Bornholt, and Vijay Chidambaram. 2023. Chipmunk: Investigating Crash-Consistency in Persistent-Memory File Systems. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 718–733. https://doi.org/10.1145/3552326.3567498

[132] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing Use-after-free with Dangling Pointers Nullification. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS '15)*. USENIX Association, 101–115. https://doi.org/10.14722/ndss.2015.23238

[133] Tanakorn Leesatapornwongsa and Haryadi S. Gunawi. 2015. SAMC: A Fast Model Checker for Finding Heisenbugs in Distributed Systems (Demo). In *Proceedings of*

the 2015 International Symposium on Software Testing and Analysis (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 423–427. https://doi.org/10.1145/2771783.2784771

[134] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014. SAMC: semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 399–414.

[135] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016*, Tom Conte and Yuanyuan Zhou (Eds.). ACM, 517–530. https://doi.org/10.1145/2872362.2872374

[136] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) (ICSE'23)*. 919–931. https://doi.org/10.1109/ICSE48619.2023.00085

[137] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 254–265.

[138] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 475–485. https://doi.org/10.1145/3238147.3238176

[139] Ao Li, Shan Lu, Suman Nath, Rohan Padhye, and Vyas Sekar. 2024. ExChain: Exception Dependency Analysis for Root Cause Diagnosis. In *Proceedings of the 21th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024 (Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024)*. USENIX Association, USA.

[140] Ao Li, Rohan Padhye, and Vyas Sekar. 2025. SPIDER: Fuzzing for Stateful Performance Issues in the ONOS Software-Defined Network Controller. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 1–12. https://doi.org/10.1109/ICST62969.2025.10988999

[141] Chi Li, Shu Wang, Henry Hoffmann, and Shan Lu. 2020. Statically inferring performance properties of software configurations. In *Proceedings of the Fifteenth Eu-*

ropean Conference on Computer Systems (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 10, 16 pages. https://doi.org/10.1145/3342195.3387520

[142] Guangpu Li, Dongjie Chen, Shan Lu, Madanlal Musuvathi, and Suman Nath. 2021. SherLock: Unsupervised Synchronization-Operation Inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 314–328. https://doi.org/10.1145/3445814.3446754

[143] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. 2019. Efficient Scalable Thread-Safety-Violation Detection: Finding Thousands of Concurrency Bugs during Testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 162–180. https://doi.org/10.1145/3341301.3359638

[144] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. Assisting Static Analysis with Large Language Models: A ChatGPT Experiment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (, San Francisco, CA, USA,) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 2107–2111. https://doi.org/10.1145/3611643.3613078

[145] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Pasadena, California, USA) *(OOPSLA'24)*. Association for Computing Machinery, New York, NY, USA.

[146] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S. Gunawi, Xiaohui Gu, Xicheng Lu, and Dongsheng Li. 2018. Pcatch: Automatically Detecting Performance Cascading Bugs in Cloud Systems. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 7, 14 pages. https://doi.org/10.1145/3190508.3190552

[147] Jiaxin Li, Yiming Zhang, Shan Lu, Haryadi S. Gunawi, Xiaohui Gu, Feng Huang, and Dongsheng Li. 2023. Performance Bug Analysis and Detection for Distributed Storage and Computing Systems. *ACM Trans. Storage* 19, 3, Article 23, 33 pages. https://doi.org/10.1145/3580281

[148] Weichen Li, Albert Jan, Baishakhi Ray, Junfeng Yang, Chengzhi Mao, and Kexin Pei. 2025. EditLord: Learning Code Transformation Rules for Code Editing. In *The Forty-Second International Conference on Machine Learning (ICML)*.

[149] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097. https://doi.org/10.1126/science.abq1158 arXiv:https://www.science.org/doi/pdf/10.1126/science.abq1158

[150] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jinsong Dong. 2017. Feedback-based debugging. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 393–403. https://doi.org/10.1109/ICSE.2017.43

[151] Bozhen Liu and Jeff Huang. 2018. D4: Fast Concurrency Debugging with Parallel Differential Analysis. (2018), 359–373. https://doi.org/10.1145/3192366.3192390

[152] Changming Liu, Deqing Zou, Peng Luo, Bin B. Zhu, and Hai Jin. 2018. A Heuristic Framework to Detect Concurrency Vulnerabilities. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) *(ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 529–541. https://doi.org/10.1145/3274694.3274718

[153] Dongge Liu, Oliver Chang, Jonathan Metzman, Martin Sablotny, and Mihai Maruseac. 2024. OSS-fuzz-gen: Automated fuzz target generation. https://github.com/google/oss-fuzz-gen

[154] Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi, and Chen Tian. 2017. DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud Systems. In *ASPLOS*.

[155] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. 2019. What bugs cause production cloud incidents?. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS'19)*. Association for Computing Machinery, New York, NY, USA, 155–162. https://doi.org/10.1145/3317550.3321438

[156] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. 2018. FCatch: Automatically Detecting Time-of-Fault Bugs in Cloud Systems. *SIGPLAN Not.* 53, 2 (March 2018), 419–431. https://doi.org/10.1145/3296957.3177161

[157] Jiawei Liu, Thanh Nguyen, Mingyue Shang, Hantian Ding, Xiaopeng Li, Yu Yu, Varun Kumar, and Zijian Wang. 2024. Learning code preference via synthetic evolution. *arXiv preprint arXiv:2410.03837* (2024).

[158] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language

Models for Code Generation. In *Advances in Neural Information Processing Systems (NeurIPS'23, Vol. 36)*, A. Oh, T. Neumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.). Curran Associates, Inc., 21558–21572. https://proceedings.neurips.cc/paper_files/paper/2023/file/43e9d647ccd3e4b7b5baab53f0368686-Paper-Conference.pdf

[159] Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. 2024. Evaluating Language Models for Efficient Code Generation. In *First Conference on Language Modeling*. https://openreview.net/forum?id=IBCBMeAhmC

[160] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2016. DoubleTake: Fast and Precise Error Detection via Evidence-Based Dynamic Analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 911–922. https://doi.org/10.1145/2884781.2884784

[161] Jie Lu, Chen Liu, Lian Li, Xiaobing Feng, Feng Tan, Jun Yang, and Liang You. 2019. CrashTuner: detecting crash-recovery bugs in cloud systems via meta-info analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP'19)*. Association for Computing Machinery, New York, NY, USA, 114–130. https://doi.org/10.1145/3341301.3359645

[162] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *ASPLOS*.

[163] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. 2015. Retro: Targeted Resource Management in Multi-tenant Distributed Systems. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 589–603. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/mace

[164] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot tracing: dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 378–393. https://doi.org/10.1145/2815400.2815415

[165] Rupak Majumdar and Filip Niksic. 2017. Why is random testing effective for partition tolerance bugs? *Proc. ACM Program. Lang.* 2, POPL, Article 46, 24 pages. https://doi.org/10.1145/3158134

[166] Paul D. Marinescu, Radu Banabic, and George Candea. 2010. An extensible technique for high-precision testing of recovery code. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA) *(USENIXATC'10)*. USENIX Association, USA, 23.

[167] Paul D. Marinescu and George Candea. 2009. LFI: A practical and general library-level fault injector. In *2009 IEEE/IFIP International Conference on Dependable Systems and Networks*. 379–388. https://doi.org/10.1109/DSN.2009.5270313

[168] Phil McMinn. 2011. Search-Based Software Testing: Past, Present and Future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. 153–163. https://doi.org/10.1109/ICSTW.2011.100

[169] Christopher S. Meiklejohn, Andrea Estrada, Yiwen Song, Heather Miller, and Rohan Padhye. 2021. Service-Level Fault Injection Testing. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC'21)*. Association for Computing Machinery, New York, NY, USA, 388–402. https://doi.org/10.1145/3472883.3487005

[170] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 33–50.

[171] Madan Musuvathi and Shaz Qadeer. 2007. CHESS: Systematic Stress Testing of Concurrent Software. *Lecture Notes in Computer Science: Logic-Based Program Synthesis and Transformation* 4407 (July 2007), 18–41.

[172] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

[173] Robert H. B. Netzer and Barton P. Miller. 1991. Improving the Accuracy of Data Race Detection. In *PPOPP*.

[174] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: detecting and fixing performance problems that have non-intrusive fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 902–912.

[175] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 562–571.

[176] Suphakit Niwattanakul, Jatsada Singthongchai, Ekkachai Naenudorn, and Supachanun Wanapu. 2013. Using of Jaccard coefficient for keywords similarity. In *Proceedings of the international multiconference of engineers and computer scientists*, Vol. 1. 380–384.

[177] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Static detection of asymptotic performance bugs in collection traversals. *SIGPLAN Not.* 50, 6 (June 2015), 369–378. https://doi.org/10.1145/2813885.2737966

[178] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576

[179] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: domain-specific fuzzing with waypoints. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 174 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360600

[180] Jia Pan, Haoze Wu, Tanakorn Leesatapornwongsa, Suman Nath, and Peng Huang. 2024. Efficient Reproduction of Fault-Induced Failures in Distributed Systems with Feedback-Driven Fault Injection. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 46–62. https://doi.org/10.1145/3694715.3695979

[181] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE) (ICSE'24)*. IEEE Computer Society, Los Alamitos, CA, USA, 866–866. https://doi.ieeecomputersociety.org/

[182] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. *SIGARCH Comput. Archit. News* 37, 1 (March 2009), 25–36. https://doi.org/10.1145/2528521.1508249

[183] L C Paulson. 1986. Natural deduction as higher-order resolution. *J. Log. Program.* 3, 3 (Oct. 1986), 237–258. https://doi.org/10.1016/0743-1066(86)90015-4

[184] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can large language models reason about program invariants?. In *Proceedings of the 40th International Conference on Machine Learning* (, Honolulu, Hawaii, USA,) *(ICML'23)*. JMLR.org, Article 1144, 25 pages.

[185] Yun Peng, Akhilesh Deepak Gotmare, Michael Lyu, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. 2024. PerfCodeGen: Improving Performance of LLM Generated Code with Execution Feedback. *arXiv preprint arXiv:2412.03578* (2024).

[186] Yun Peng, Jun Wan, Yichen Li, and Xiaoxue Ren. 2025. COFFE: A Code Efficiency Benchmark for Code Generation. arXiv:2502.02827 [cs.SE] https://arxiv.org/abs/2502.02827

[187] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2155–2168. https://doi.org/10.1145/3133956.3134073

[188] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2155–2168.

[189] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. All file systems are not created equal: on the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 433–448.

[190] AspectJ Maven Plugin. Accessed: April 2024. https://www.mojohaus.org/aspectj-maven-plugin/.

[191] Eli Pozniansky and Assaf Schuster. 2007. MultiRace: efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience* 19, 3 (2007), 327–340. https://doi.org/10.1002/cpe.1064 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1064

[192] Michael Pradel and Thomas R. Gross. 2012. Fully Automatic and Precise Detection of Thread Safety Violations. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 521–530. https://doi.org/10.1145/2254064.2254126

[193] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2011. LOCKSMITH: Practical Static Race Detection for C. *ACM Trans. Program. Lang. Syst.* 33, 1, Article 3 (jan 2011), 55 pages. https://doi.org/10.1145/1889997.1890000

[194] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. 2009. Error propagation analysis for file systems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI'09)*. Association for Computing Machinery, New York, NY, USA, 270–280. https://doi.org/10.1145/1542476.1542506

[195] Malavika Samak and Murali Krishna Ramanathan. 2014. Multithreaded Test Synthesis for Deadlock Detection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 473–489. https://doi.org/10.1145/2660193.2660238

[196] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *SOSP*.

[197] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wör-ner, and Thorsten Holz. 2021. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2597–2614. https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo

[198] Koushik Sen. 2008. Race Directed Random Testing of Concurrent Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 11–21. https://doi.org/10.1145/1375581.1375584

[199] Kostya Serebryany. 2017. OSS-Fuzz - Google's continuous fuzzing service for open source software. USENIX Association, Vancouver, BC.

[200] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, USA, 28.

[201] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (New York, New York, USA) *(WBIA '09)*. Association for Computing Machinery, New York, NY, USA, 62–71. https://doi.org/10.1145/1791194.1791203

[202] Utsav Sethi, Haochen Pan, Shan Lu, Madanlal Musuvathi, and Suman Nath. 2022. Cancellation in Systems: An Empirical Study of Task Cancellation Patterns and Failures. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*. USENIX Association, Carlsbad, CA, 127–141. https://www.usenix.org/conference/osdi22/presentation/sethi

[203] Wenxuan Shi, Yunhang Zhang, Xinyu Xing, and Jun Xu. 2024. Harnessing Large Language Models for Seed Generation in Greybox Fuzzing. *arXiv preprint arXiv:2411.18143* (2024).

[204] Yao Shi, Soyeon Park, Zuoning Yin, Shan Lu, Yuanyuan Zhou, Wenguang Chen, and Weimin Zheng. 2010. Do I Use the Wrong Definition? DeFuse: Definition-Use

Invariants for Detecting Concurrency and Sequential Bugs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. Association for Computing Machinery, New York, NY, USA, 160–174. https://doi.org/10.1145/1869459.1869474

[205] Alexander G Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R. Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2024. Learning Performance-Improving Code Edits. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=ix7rLVHXyY

[206] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google, Inc. http://research.google.com/archive/papers/dapper-2010-1.pdf

[207] Margarita Simonova. [n. d.]. Costly Code: The Price Of Software Errors. *Forbes Magazine* ([n. d.]). https://www.forbes.com/councils/forbestechcouncil/2023/12/26/costly-code-the-price-of-software-errors/

[208] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. 2019. PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary. In *Network and Distributed System Security Symposium (NDSS)*.

[209] Prashast Srivastava and Mathias Payer. 2021. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th acm sigsoft international symposium on software testing and analysis*. 244–256.

[210] Bogdan Alexandru Stoica, Shan Lu, Madanlal Musuvathi, and Suman Nath. 2023. WAFFLE: Exposing Memory Ordering Bugs Efficiently with Active Delay Injection. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 111–126. https://doi.org/10.1145/3552326.3567507

[211] Bogdan Alexandru Stoica*, Utsav Sethi*, Yiming Su, Cyrus Zhou, Shan Lu, Jonathan Mace, Madan Musuvathi, and Suman Nath (*equal contribution). 2024. If At First You Don't Succeed, Try, Try, Again...? Insights and LLM-informed Tooling for Detecting Retry Bugs in Software Systems. In *Proceedings of the 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*.

[212] Cesar A. Stuardo, Tanakorn Leesatapornwongsa, Riza O. Suminto, Huan Ke, Jeffrey F. Lukman, Wei-Chiu Chuang, Shan Lu, and Haryadi S. Gunawi. 2019. Scalecheck: a single-machine approach for discovering scalability bugs in large distributed systems. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies* (Boston, MA, USA) *(FAST'19)*. USENIX Association, USA, 359–373.

[213] Yiming Su, Chengcheng Wan, Utsav Sethi, Shan Lu, Madan Musuvathi, and Suman Nath. 2023. HotGPT: How to Make Software Documentation More Useful with a Large Language Model?. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems* (Providence, RI, USA) *(HOTOS'23)*. Association for Computing Machinery, New York, NY, USA, 87–93. https://doi.org/10.1145/3593856.3595910

[214] Valgrind's Memchecker Sub-system. Retrieved July 1, 2025. https://valgrind.org/docs/manual/mc-manual.html.

[215] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. HEALER: Relation Learning Guided Kernel Fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 344–358. https://doi.org/10.1145/3477132.3483547

[216] Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. 2022. Automatic Reliability Testing For Cluster Management Controllers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 143–159. https://www.usenix.org/conference/osdi22/presentation/sun

[217] Lilia Tang, Chaitanya Bhandari, Yongle Zhang, Anna Karanika, Shuyang Ji, Indranil Gupta, and Tianyin Xu. 2023. Fail through the Cracks: Cross-System Interaction Failures in Modern Cloud Systems. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys'23)*. Association for Computing Machinery, New York, NY, USA, 433–451. https://doi.org/10.1145/3552326.3587448

[218] Class `CallContext`. Retrieved October 4, 2022. (Retrieved October 4, 2022). https://docs.microsoft.com/en-us/dotnet/api/system.runtime.remoting.messaging.callcontext.

[219] Class `InheritableThreadLocal`. Retrieved October 4, 2022. (Retrieved October 4, 2022). https://docs.oracle.com/javase/8/docs/api/java/lang/InheritableThreadLocal.html.

[220] The ImmunEFI Tool. September 2021. Bitswift race condition bug fix postmortem. (September 2021). https://medium.com/immunefi/bitswift-race-condition-bug-fix-postmortem-588184b8b43e.

[221] Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. 2022. Fuzzing Hardware Like Software. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3237–3254. https://www.usenix.org/conference/usenixsecurity22/presentation/trippel

[222] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. DangSan: Scalable Use-after-Free Detection. In *Proceedings of the Twelfth European Conference on Com-*

puter Systems (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 405–419. https://doi.org/10.1145/3064176.3064211

[223] Vasudev Vikram, Rohan Padhye, and Koushik Sen. 2021. Growing a Test Corpus with Bonsai Fuzzing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 723–735. https://doi.org/10.1109/ICSE43902.2021.00072

[224] Laurel Wamsley. [n. d.]. Boeing CEO Faces Tough Questions On 737 Max Plane's Design. *National Public Radio (NPR)* ([n. d.]). https://www.npr.org/2019/04/29/718283719/boeing-ceo-faces-tough-questions-on-737-max-planes-design

[225] The WASABI Toolkit. Release: September 2024. https://github.com/bastoica/wasabi.

[226] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. 2022. Chain of Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (Eds.). https://openreview.net/forum?id=_VjQlMeSB_J

[227] Brian Wickman, Hong Hu, Insu Yun, DaeHee Jang, JungWon Lim, Sanidhya Kashyap, and Taesoo Kim. 2021. Preventing Use-After-Free Attacks with Fast Forward Allocation. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2453–2470. https://www.usenix.org/conference/usenixsecurity21/presentation/wickman

[228] Wikipedia. 2025. Diophantine equation — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Diophantine%20equation&oldid=1290376023. [Online; accessed 15-May-2025].

[229] Haoze Wu, Jia Pan, and Peng Huang. 2024. Efficient Exposure of Partial Failure Bugs in Distributed Systems with Inferred Abstract States. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 1267–1283. https://www.usenix.org/conference/nsdi24/presentation/wu-haoze

[230] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) (ICSE'23)*. 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129

[231] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 2009. Go with the flow: profiling copies to find runtime bloat. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 419–430. https://doi.org/10.1145/1542476.1542523

[232] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2010. Finding low-utility data structures. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 174–186. https://doi.org/10.1145/1806596.1806617

[233] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data Race Fuzzing for Kernel File Systems. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1643–1660. https://doi.org/10.1109/SP40000.2020.00078

[234] Holly Yan. [n. d.]. A "dangerously unacceptable breakdown" led to errant or delayed evacuation warnings in LA. The rest of the US isn't immune. *Cable News Network (CNN)* ([n. d.]). https://www.cnn.com/2025/01/16/us/evacuation-warnings-vulnerabilities-la-fires

[235] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2018. Spatio-Temporal Context Reduction: A Pointer-Analysis-Based Static Approach for Detecting Use-after-Free Vulnerabilities *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 327–337. https://doi.org/10.1145/3180155.3180178

[236] Jun Yang, Cheng-Chi Wang, Bogdan Alexandru Stoica, and Kexin Pei. 2025. Synthesizing Performance Constraints for Evaluating and Improving Code Efficiency. arXiv:2505.23471 [cs.SE] https://arxiv.org/abs/2505.23471

[237] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. Simple testing can prevent most critical failures: an analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 249–265.

[238] Michał Zalewski. Retrieved July 1, 2025. American Fuzzy Loop. https://lcamtuf.coredump.cx/afl/.

[239] Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. 2024. Self-taught optimizer (stop): Recursively self-improving code generation. In *First Conference on Language Modeling*.

[240] Sheng Zhan and Jeff Huang. 2016. ECHO: Instantaneous in Situ Race Detection in the IDE. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 775–786. https://doi.org/10.1145/2950290.2950332

[241] Cen Zhang, Yaowen Zheng, Mingqiang Bai, Yeting Li, Wei Ma, Xiaofei Xie, Yuekang Li, Limin Sun, and Yang Liu. 2024. How effective are they? Exploring large language model based fuzz driver generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1223–1235.

[242] Jiyang Zhang, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2023. Multilingual Code Co-evolution using Large Language Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (, San Francisco, CA, USA,) *(ESEC/FSE'23)*. Association for Computing Machinery, New York, NY, USA, 695–707. https://doi.org/10.1145/3611643.3616350

[243] Pingyu Zhang and Sebastian Elbaum. 2012. Amplifying tests to validate exception handling code. In *2012 34th International Conference on Software Engineering (ICSE) (ICSE'12)*. 595–605. https://doi.org/10.1109/ICSE.2012.6227157

[244] Pingyu Zhang, Sebastian Elbaum, and Matthew B. Dwyer. 2011. Automatic generation of load tests. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. 43–52. https://doi.org/10.1109/ASE.2011.6100093

[245] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas W. Reps. 2011. ConSeq: detecting concurrency bugs through sequential errors. In *ASPLOS*.

[246] Wei Zhang, Chong Sun, and Shan Lu. 2010. ConMem: Detecting Severe Concurrency Bugs through an Effect-Oriented Approach. *SIGPLAN Not.* 45, 3, 179–192. https://doi.org/10.1145/1735971.1736041

[247] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 379–391. https://doi.org/10.1145/2465351.2465388

[248] Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Kirk Rodrigues, Shan Lu, and Ding Yuan. 2021. Understanding and Detecting Software Upgrade Failures in Distributed Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 116–131. https://doi.org/10.1145/3477132.3483577

[249] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 5673–5684.